ATL and ActiveX Controls

Program examples compiled using Visual C++ 6.0 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 control class.

Index:

If you've finished reading about COM and ATL and still wonder how COM fits into your day-to-day programming activities, you're not alone. Figuring out how to use COM in real life isn't always obvious at first glance. After all, a whole lot of extra code must be typed in just to get a COM object up and running. However, there's a very **real application** of COM right under your nose, **ActiveX Controls**. ActiveX controls are small gadgets (usually UI-oriented) written around the Component Object Model.
In Module 28, you examined COM classes created by using ATL. In this module, you'll learn how to write a certain kind of COM class, an ActiveX control. You had a chance to work with ActiveX Controls from the client side in Module 18. Now it's time to write your own. There are several steps involved in creating an ActiveX control using ATL, including:

- Deciding what to draw.
- Developing incoming interfaces for the control.
- Developing outgoing interfaces (events) for the control.
- Implementing a persistence mechanism for the control.
- Providing a user interface for manipulating the control's properties.

This module covers all these steps. Soon, you'll be able to use ATL to create ActiveX controls that you (or other developers) can use within other programs. The next two modules will present ActiveX control program examples compiled using Visual C++ .Net.

ActiveX Controls

Even today, there's some confusion as to what really constitutes an ActiveX control. In 1994, Microsoft tacked some new interfaces onto its Object Linking and Embedding protocol, packaged them within DLLs, and called them OLE Controls. Originally, **OLE Controls** implemented nearly the entire OLE Document embedding protocol. In addition, OLE Controls supported the following:

- Dynamic invocation (Automation).
- Property pages (so the user could modify the control's properties).

- Outbound callback interfaces (event sets).
- Connections (a standard way to for clients and controls to hook up the event callbacks).

When the Internet became a predominant factor in Microsoft's marketing plans, Microsoft announced its intention to plant ActiveX Controls on Web pages. At that point, the size of these components became an issue. Microsoft took its OLE Control specification, changed the name from **OLE Controls** to **ActiveX Controls**, and stated that all the features listed above were optional. This means that under the new ActiveX Control definition, a control's only requirement is that it be based on COM and that it implements `IUnknown`. Of course, for a control to be useful it really needs to implement most of the features listed above. So in the end, ActiveX Controls and OLE Controls refer to more or less the same animal.
Developers have been able to use MFC to create ActiveX controls since mid-1994. However, one of the downsides of using MFC to create ActiveX controls is that the controls become bound to MFC. Sometimes you want your controls to be smaller or to work even if the end user doesn't have the MFC DLLs on his or her system. In addition, using MFC to create ActiveX controls forces you into making certain design decisions. For example, if you decide to use MFC to write an ActiveX control, you more or less lock yourself out of using dual interfaces (unless you feel like writing a lot of extra code). Using MFC to create ActiveX controls also means the control and its property pages need to use IDispatch to communicate among themselves.
To avoid the problems described so far, developers can now use ATL to create ActiveX controls. ATL now includes the facilities to create full-fledged ActiveX controls, complete with every feature an ActiveX control should have. These features include incoming interfaces, persistent properties, property pages, and connection points. If you've ever written an ActiveX control using MFC, you'll see how much more flexible using ATL can be.

Using ATL to Write a Control

Although creating an ActiveX control using ATL is actually a pretty straightforward process, using ATL ends up being a bit more burdensome than using MFC. That's because ATL doesn't include all of MFC's amenities. For example, ATL doesn't include device context wrappers. When you draw on a device context, you need to use the raw device context handle. In addition, ClassWizard doesn't understand ATL-based source code, so when you want your control to handle messages, you end up using the "TypingWizard". That is, you end up typing the message maps in by hand.
Despite these issues, creating an ActiveX control using ATL is a whole lot easier than creating one from scratch. Also, using ATL gives you a certain amount of flexibility you don't get when you use MFC. For example, while adding dual interfaces to your control is a tedious process with MFC, you get them for free when you use ATL. The **ATL COM Object Wizard** also makes adding more COM classes (even non-control classes) to your project very easy, while adding new controls to an MFC-based DLL is a bit more difficult.
For this module's example, we'll represent a small pair of dice as an ATL-based ActiveX control. The dice control will illustrate the most important facets of ActiveX Controls, including control rendering, incoming interfaces, properties, property pages, and events.

The Myatldicesvr Program From Scratch

Before we dig into the story, let build an ATL program from scratch. As usual, launch AppWizard by clicking **File New** menu of the Visual C++. Then, just follow the shown steps.

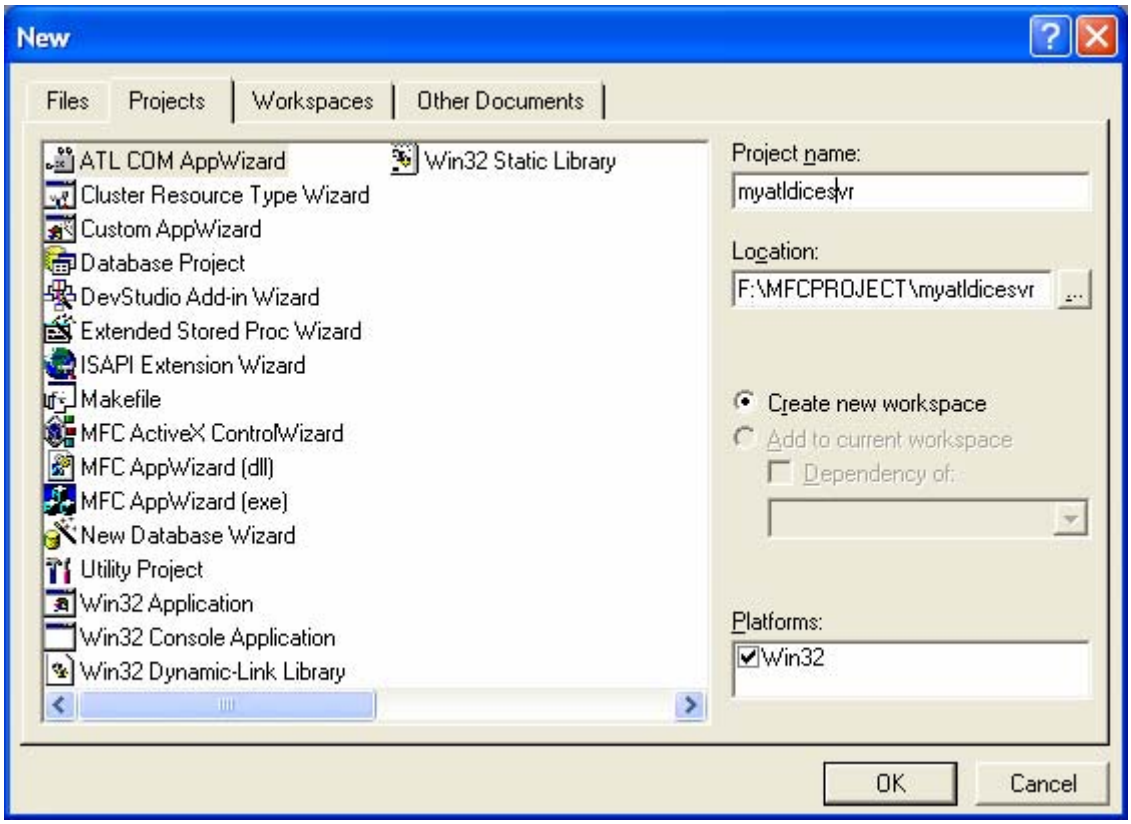

Figure 1: **Myatldicesvr** - Visual C++ new ATL COM AppWizard project dialog.

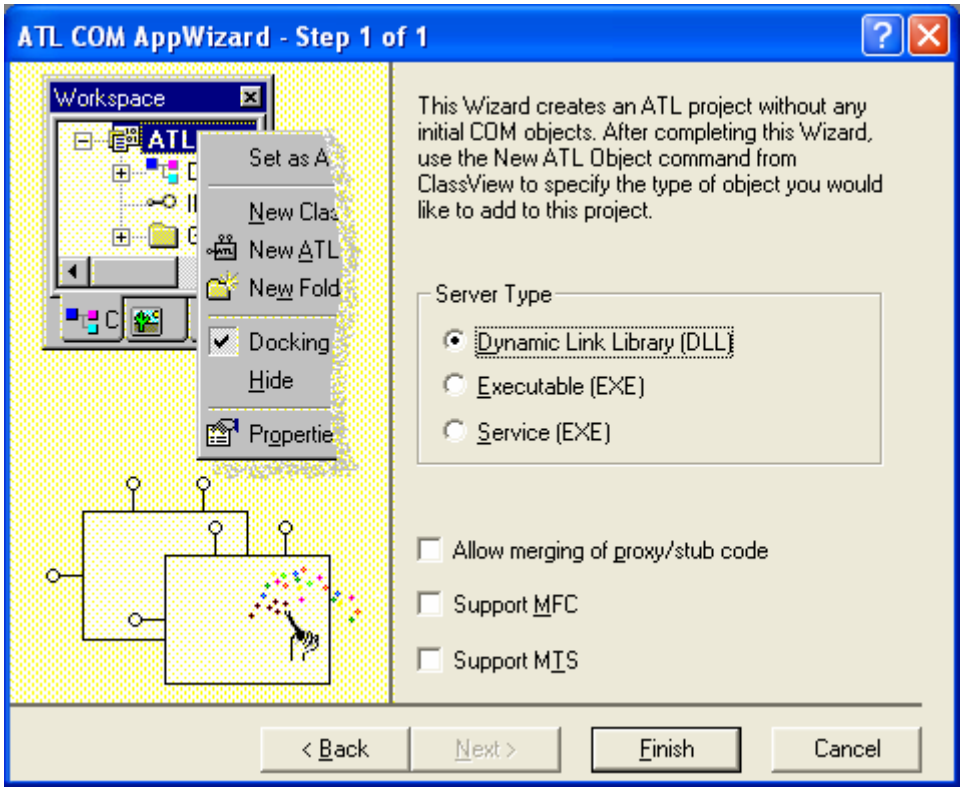

Figure 2: ATL COM AppWizard step 1 of 1.

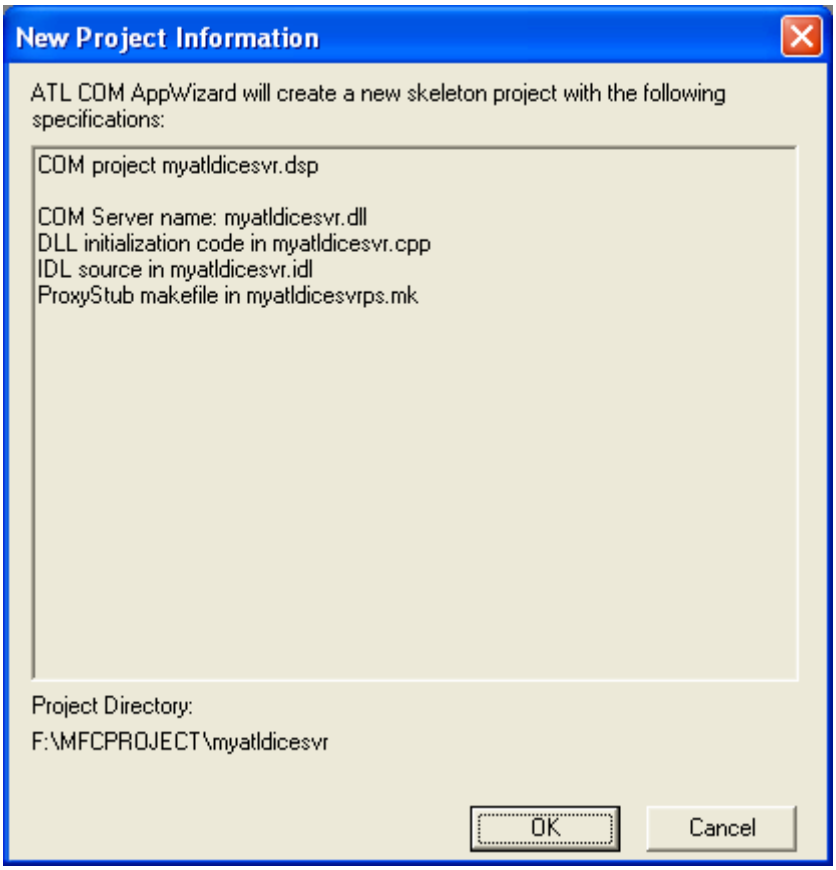

Figure 3: **Myatldicesvr** project summary.

Add new ATL object.

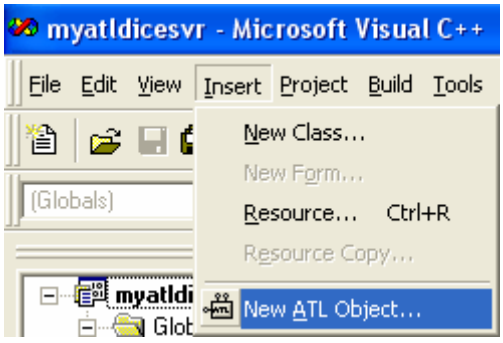

Figure 4: Adding new ATL object.

Select **Controls** in **Category** list and **Full Control** in **Objects** list. Then click the **Next** button.

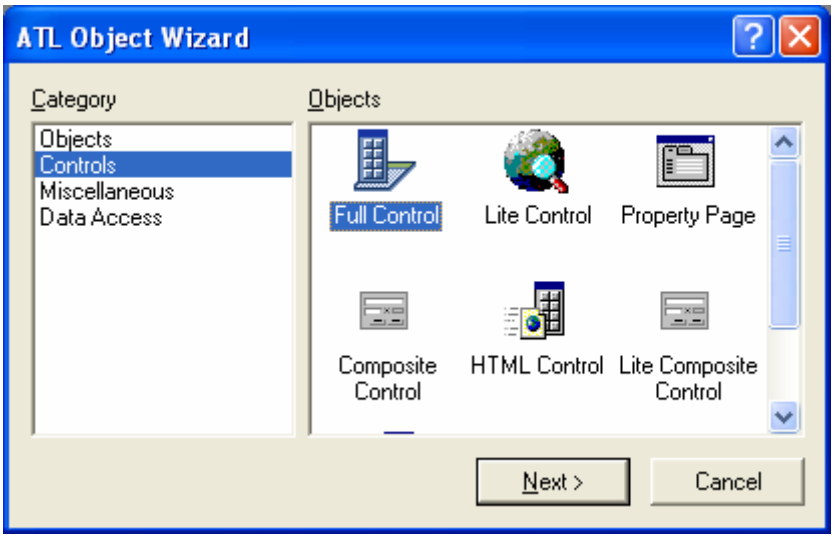

Figure 5: ATL object wizard, adding **Full Control** object.

Type in **myatldiceob** as object name, others will be automatically provided. You can change as needed.

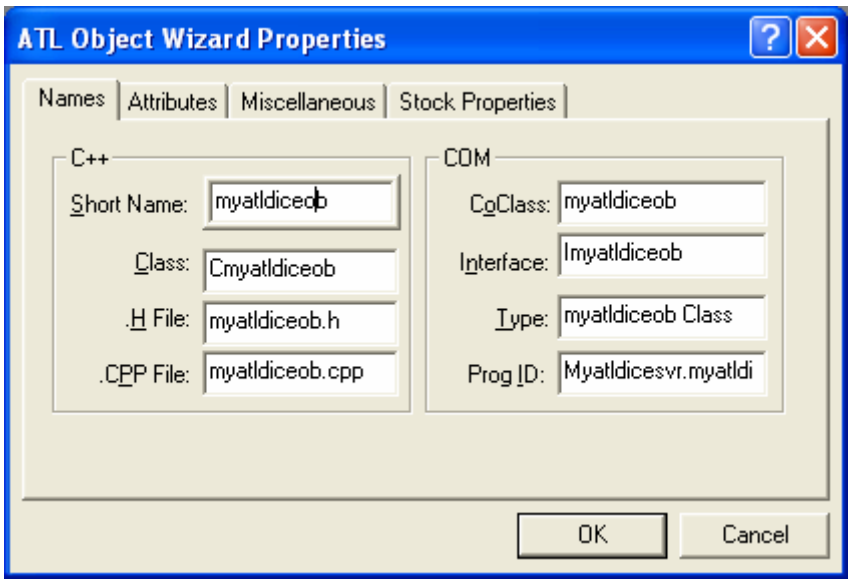

Figure 6: **Myatldicesvr**'s object name.

Tick the **Support Connection Points** for **Attributes**.

Figure 7: Adding the **Support Connection Point** attribute to ATL object.

Just accept the default for **Miscellaneous**.

Figure 8: Accepting default miscellaneous options.

We select the **Background Color** for the **Stock Properties**.

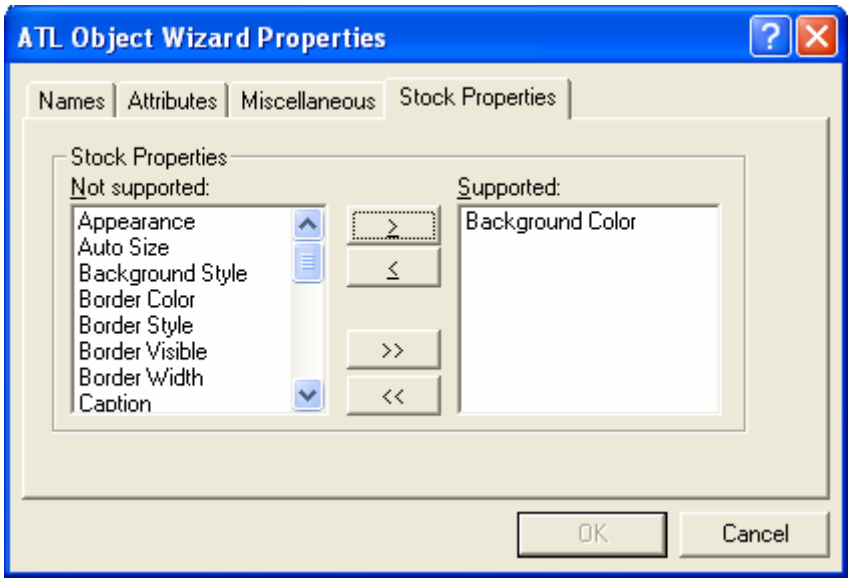

Figure 9: Adding background of the Stock Properties.

Start creating bitmaps for white, blue and red colors. Use the **Copy** and **Paste** menu under the **Edit** to speed up your work :-).

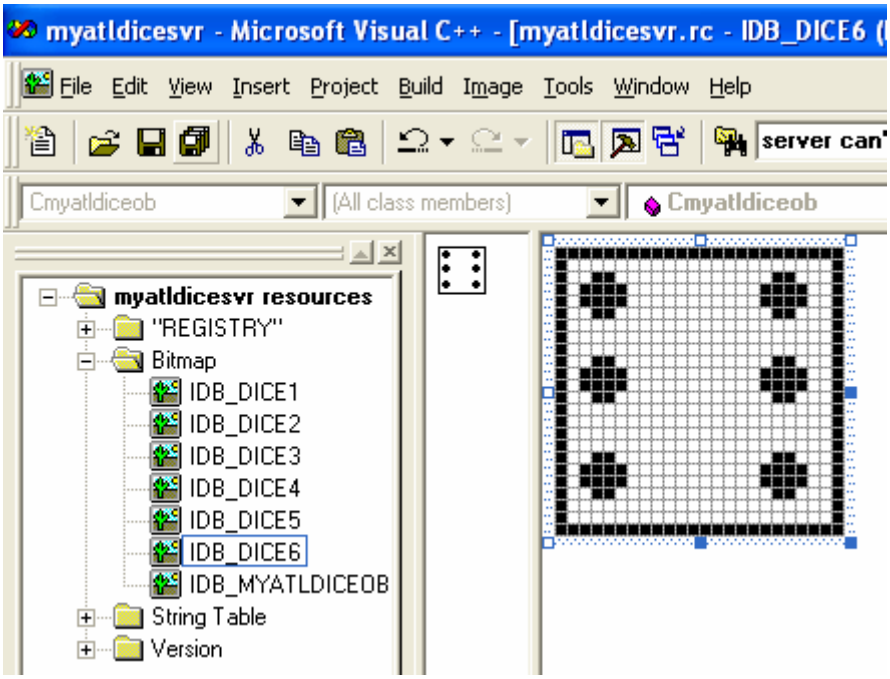

Figure 10: Adding white dice bitmaps.

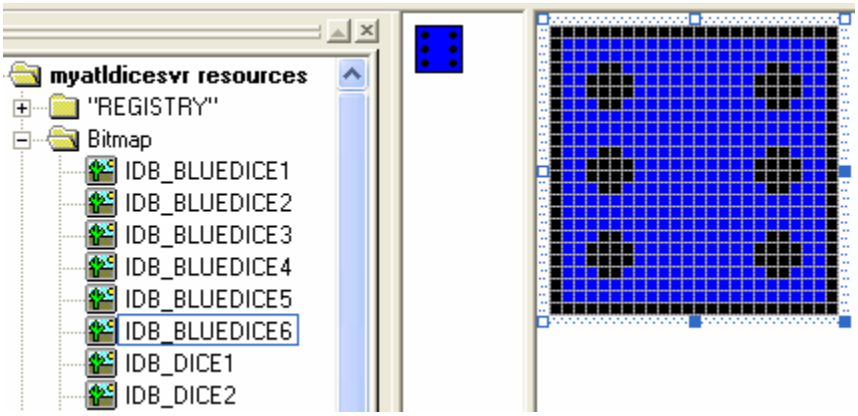

Figure 11: Adding blue dice bitmaps.

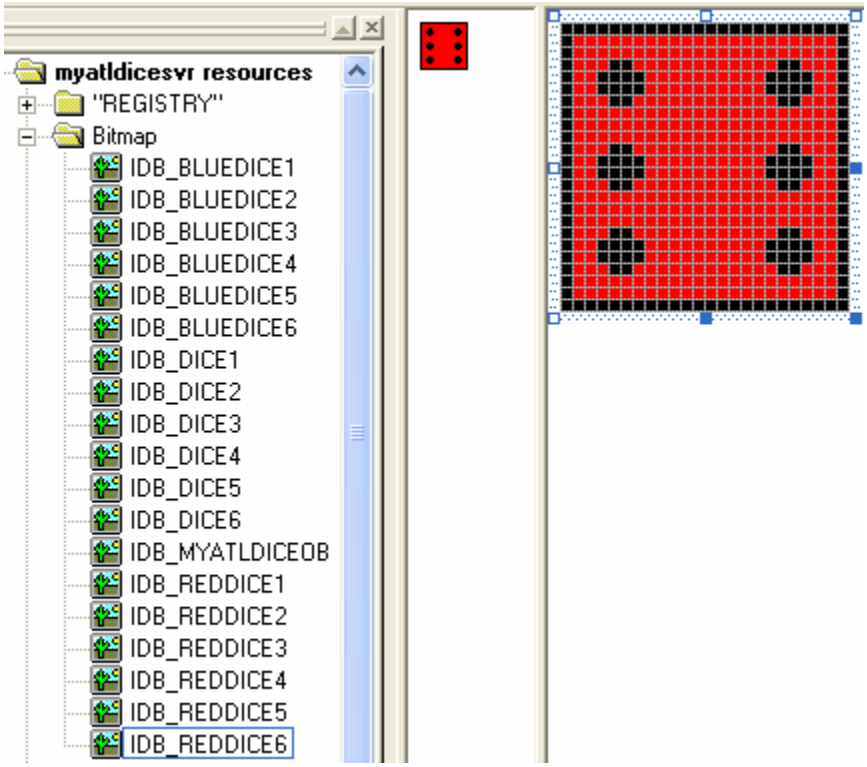

Figure 12: Adding red dice bitmaps.

Now we can start the coding part. Put codes in **myatldiceob.h**. Firstly add the `MAX_DIEFACES` constant.

```
#define MAX_DIEFACES 6
```

```
#ifndef __MYATLDICEOB_H_
#define __MYATLDICEOB_H_

#include "resource.h"
#include <atlctl.h>

#define MAX_DIEFACES 6

////////////////////////////
```

Listing 1.

Using ClassView, add `LoadBitmaps()` function to `Cmyatldiceob` class as shown below.

```
BOOL LoadBitmaps();
```

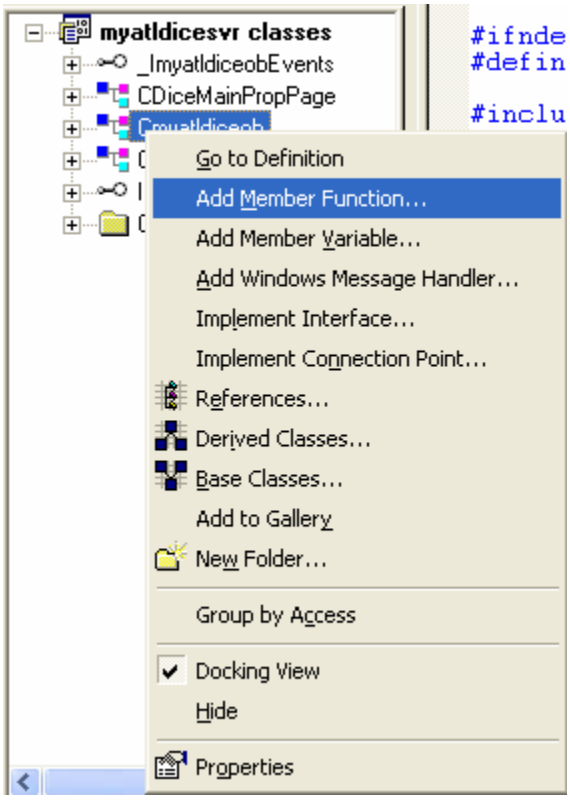

Figure 13: Adding functions and variables to `Cmyatldiceob` class.

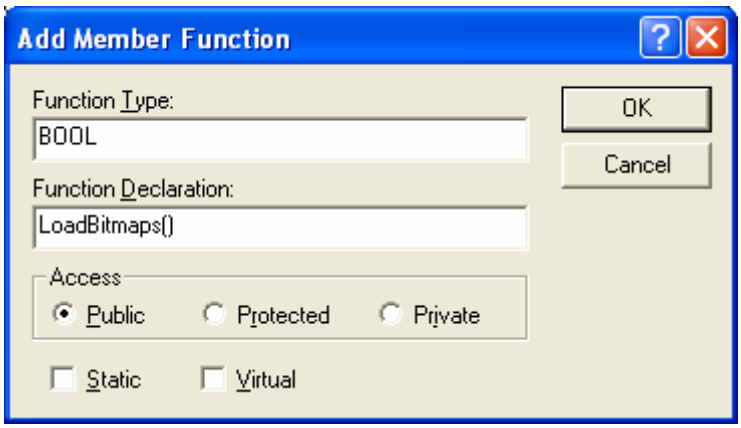

Figure 14: Adding `LoadBitmap()` function.

Add the `HBITMAP m_dieBitmaps[MAX_DIEFACES]` array variable.

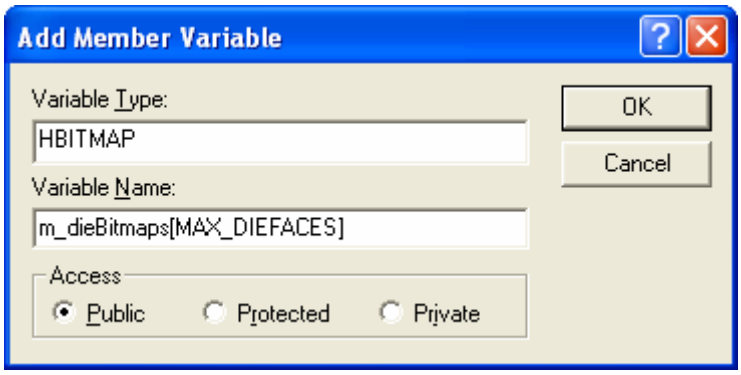

Figure 15: Adding an array variable.

Using ClassView, add the following member functions.

```
void ShowFirstDieFace(ATL_DRAWINFO& di);
void ShowSecondDieFace(ATL_DRAWINFO& di);
```

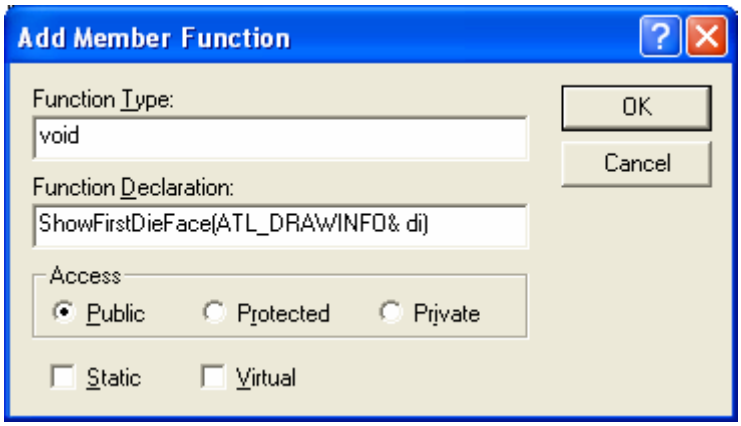

Figure 16: Adding `ShowFirstDieFace()` member function.

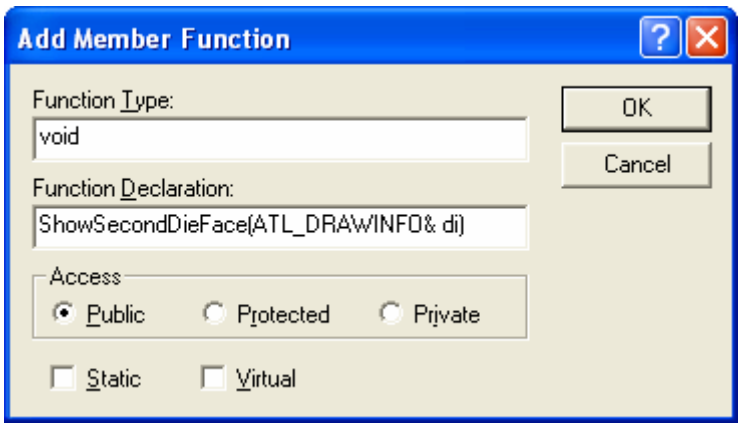

Figure 17: Adding `ShowSecondDieFace()` function to `Cmyatldiceob` class.

Using ClassView, add the following member variables.

```
short m_nDiceColor;
short m_nTimesToRoll;
short m_nTimesRolled;
```

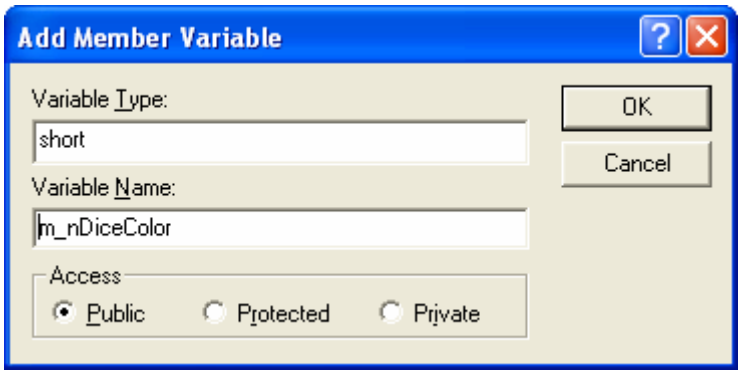

Figure 18: Adding member variables to `Cmyatldiceob` class.

Add the following member variables.

```
unsigned short m_nFirstDieValue;
unsigned short m_nSecondDieValue;
```

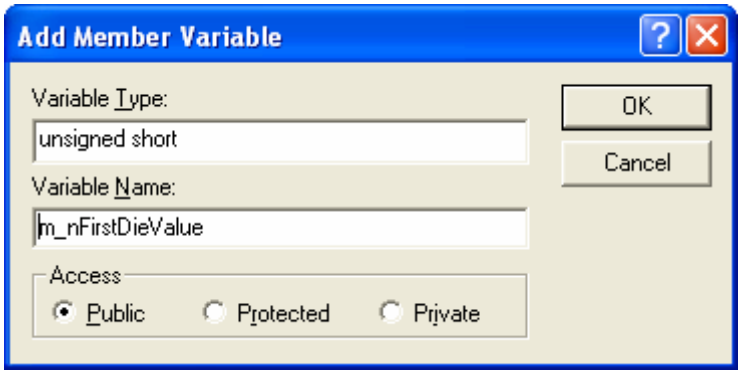

Figure 19: Adding member variable, `m_nFirstDieValue` to `Cmyatldiceob` class.

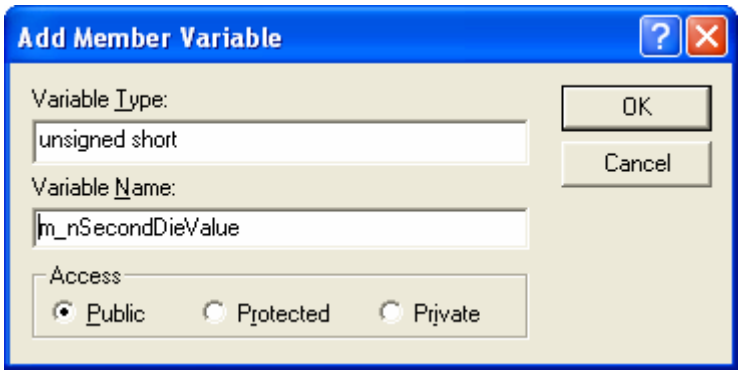

Figure 20: Adding member variable, `m_nSecondDieValue` to `Cmyatldiceob` class.

The following are the previously added member functions and variables. Take note that, the codes have been relocated just after the `END_MSG_MAP()`.

```
//  LRESULT NotifyHandler(int idCtrl, LPNMHDR

    unsigned short m_nSecondDieValue;
    unsigned short m_nFirstDieValue;
    short m_nTimesRolled;
    short m_nTimesToRoll;
    short m_nDiceColor;
    void ShowSecondDieFace(ATL_DRAWINFO& di);
    void ShowFirstDieFace(ATL_DRAWINFO& di);
    HBITMAP m_dieBitmaps[MAX_DIEFACES];
    BOOL LoadBitmaps();

// IViewObjectEx
```

Listing 2.

Add the `Cmyatldiceob()` codes. These codes are variables initialization.

```
Cmyatldiceob()
      {
            m_bWindowOnly = TRUE;

            LoadBitmaps();
            srand((unsigned)time(NULL));
            m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
            m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;

            m_nTimesToRoll = 15;
            m_nTimesRolled = 0;
            m_nDiceColor = 0;
      }
```

```
public:
    Cmyatldiceob()
    {
        m_bWindowOnly = TRUE;

        LoadBitmaps();
        srand((unsigned)time(NULL));
        m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
        m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;

        m_nTimesToRoll = 15;
        m_nTimesRolled = 0;
        m_nDiceColor = 0;
    }
```

Listing 3.

Add the `#include` directive for time related function.

```
#include "time.h"
```

```
#ifndef __MYATLDICEOB_H_
#define __MYATLDICEOB_H_

#include "resource.h"
#include <atlctl.h>
#include "time.h"

#define MAX_DIEFACES 6
```

Listing 4.

Edit the `OnDraw()`. Take note that, here, you will find that the location of the `OnDraw()` and a few other functions declared and defined in **myatldiceob.h** (default if using ClassView) while in the story part of this Tutorial, the functions declared in **myatldiceob.h** and defined in **myatldiceob.cpp**.

```
HRESULT OnDraw(ATL_DRAWINFO& di)
{
	RECT& rc = *(RECT*)di.prcBounds;
	ShowFirstDieFace(di);
	ShowSecondDieFace(di);
	return S_OK;
}
```

```
// Imyatldiceob
public:

	HRESULT OnDraw(ATL_DRAWINFO& di)
	{
		RECT& rc = *(RECT*)di.prcBounds;
		ShowFirstDieFace(di);
		ShowSecondDieFace(di);
		return S_OK;
	}

	OLE_COLOR m_clrBackColor;
};

#endif //__MYATLDICEOB_H_
```

Listing 5.

Add `WM_TIMER` (`OnTimer()`) Windows message handler using ClassView to `Cmyatldiceob` class.

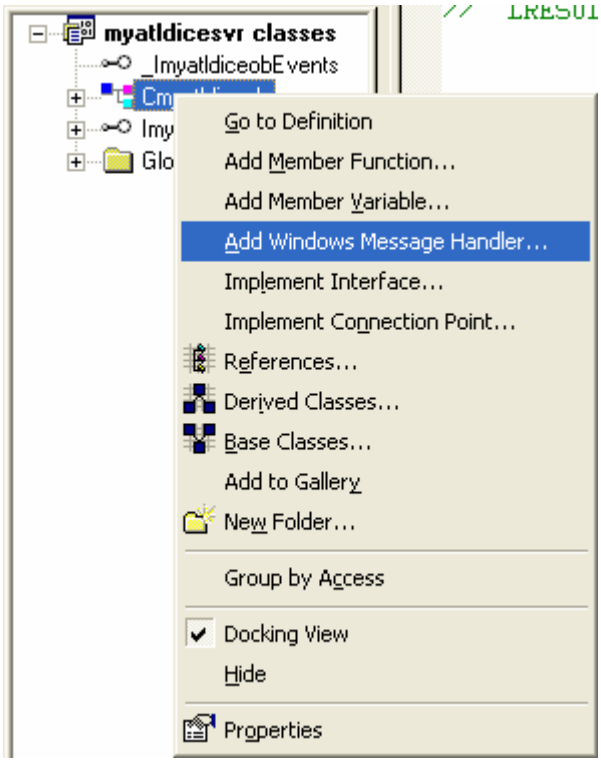

Figure 21: Adding `WM_TIMER` Windows message handler to `Cmyatldiceob` class.

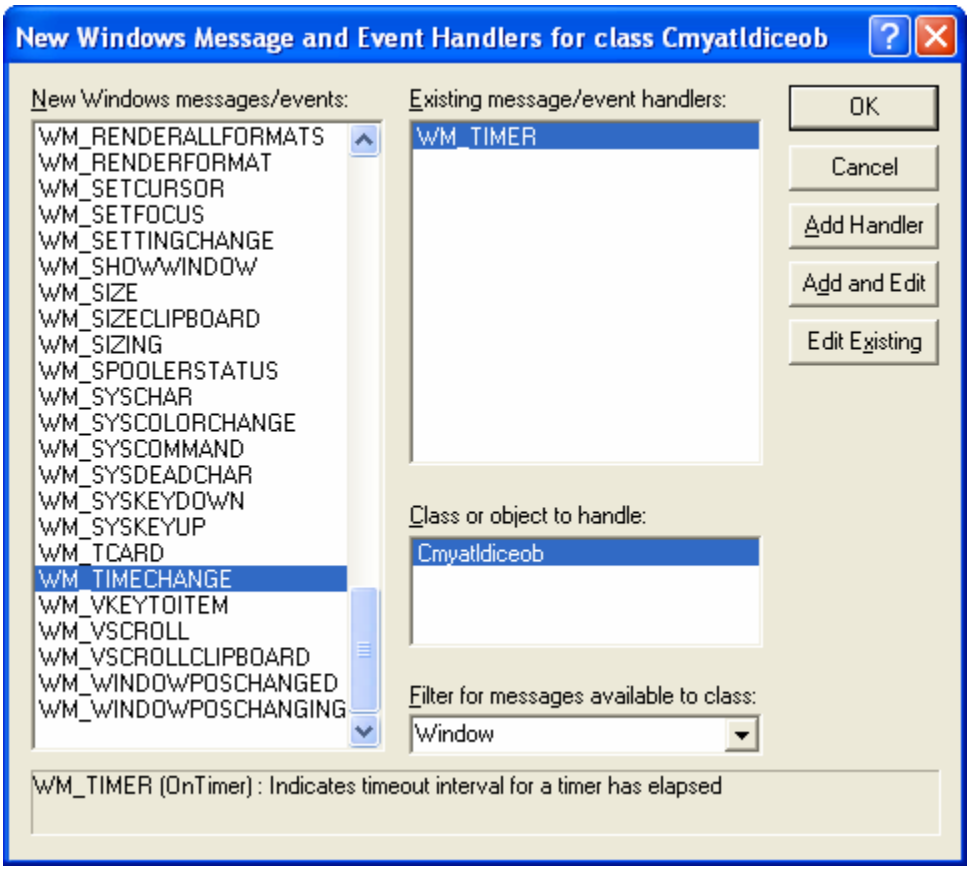

Figure 22: Adding `WM_TIMECHANGE` Windows message handler.

Add the `OnTimer()` code.

```
LRESULT OnTimer(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
	// TODO : Add Code for message handler.
	// Call DefWindowProc if necessary.
	if(m_nTimesRolled > 15)
	{
		m_nTimesRolled = 0;
		KillTimer(1);
	} else {
		m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
		m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
		FireViewChange();
		m_nTimesRolled++;
	}
	bHandled = TRUE;
	return 0;
}
```

```
LRESULT OnTimer(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
    // TODO : Add Code for message handler. Call DefWindowProc if necessary.
    if(m_nTimesRolled > 15)
    {
        m_nTimesRolled = 0;
        KillTimer(1);
    } else {
        m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
        m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
        FireViewChange();
        m_nTimesRolled++;
    }
    bHandled = TRUE;
    return 0;
    }
```

Listing 6.

Add other implementation codes in **myatldiceob.cpp** for `LoadBitmaps()`, `ShowFirstDieFace()` and `ShowSecondDieFace()`.

```
BOOL Cmyatldiceob::LoadBitmaps()
{
      int i;
      BOOL bSuccess = TRUE;
      int nID = IDB_DICE1;

      for(i=0; i<MAX_DIEFACES; i++)
    {
      DeleteObject(m_dieBitmaps[i]);
      m_dieBitmaps[i] = LoadBitmap(_Module.m_hInst, MAKEINTRESOURCE(nID+i));
      if(!m_dieBitmaps[i])
        {
          ::MessageBox(NULL,
                       "Failed to load bitmaps",
                       NULL,
                       MB_OK);
          bSuccess = FALSE;
      }
   }
   OutputDebugString("Got to the LoadBitmaps functions\n");
   return bSuccess;
}
```

```
BOOL Cmyatldiceob::LoadBitmaps()
{
    int i;
    BOOL bSuccess = TRUE;
    int nID = IDB_DICE1;

    for(i=0; i<MAX_DIEFACES; i++)
      {
      DeleteObject(m_dieBitmaps[i]);
      m_dieBitmaps[i] = LoadBitmap(_Module.m_hInst, MAKEINTRESOURCE(nID+i));
      if(!m_dieBitmaps[i])
      {
          ::MessageBox(NULL,
                       "Failed to load bitmaps",
                       NULL,
                       MB_OK);
          bSuccess = FALSE;
      }
   }
   OutputDebugString("Got to the LoadBitmaps functions\n");
   return bSuccess;
}
```

Listing 7.

```
void Cmyatldiceob::ShowFirstDieFace(ATL_DRAWINFO &di)
{
      BITMAP bmInfo;
      GetObject(m_dieBitmaps[m_nFirstDieValue-1],
         sizeof(bmInfo), &bmInfo);

      SIZE size;

      size.cx = bmInfo.bmWidth;
      size.cy = bmInfo.bmHeight;

      HDC hMemDC;
      hMemDC = CreateCompatibleDC(di.hdcDraw);

      HBITMAP hOldBitmap;
      HBITMAP hbm = m_dieBitmaps[m_nFirstDieValue-1];
      hOldBitmap = (HBITMAP)SelectObject(hMemDC, hbm);

      if (hOldBitmap == NULL)
      return;     // destructors will clean up

      BitBlt(di.hdcDraw,
          di.prcBounds->left+1,
          di.prcBounds->top+1,
          size.cx,
          size.cy,
          hMemDC, 0,
          0,
          SRCCOPY);

      SelectObject(di.hdcDraw, hOldBitmap);
      DeleteDC(hMemDC);
}
```

```
void Cmyatldiceob::ShowFirstDieFace(ATL_DRAWINFO &di)
{
    BITMAP bmInfo;
    GetObject(m_dieBitmaps[m_nFirstDieValue-1],
       sizeof(bmInfo), &bmInfo);

    SIZE size;

    size.cx = bmInfo.bmWidth;
    size.cy = bmInfo.bmHeight;

    HDC hMemDC;
    hMemDC = CreateCompatibleDC(di.hdcDraw);

    HBITMAP hOldBitmap;
    HBITMAP hbm = m_dieBitmaps[m_nFirstDieValue-1];
    hOldBitmap = (HBITMAP)SelectObject(hMemDC, hbm);

    if (hOldBitmap == NULL)
      return;     // destructors will clean up

    BitBlt(di.hdcDraw,
          di.prcBounds->left+1,
          di.prcBounds->top+1,
          size.cx,
          size.cy,
          hMemDC, 0,
          0,
          SRCCOPY);

    SelectObject(di.hdcDraw, hOldBitmap);
    DeleteDC(hMemDC);
}
```

Listing 8.

```
void Cmyatldiceob::ShowSecondDieFace(ATL_DRAWINFO &di)
{
      BITMAP bmInfo;
      GetObject(m_dieBitmaps[m_nFirstDieValue-1],
         sizeof(bmInfo), &bmInfo);

      SIZE size;

      size.cx = bmInfo.bmWidth;
      size.cy = bmInfo.bmHeight;

      HDC hMemDC;
      hMemDC = CreateCompatibleDC(di.hdcDraw);

      HBITMAP hOldBitmap;
      HBITMAP hbm = m_dieBitmaps[m_nSecondDieValue-1];
      hOldBitmap = (HBITMAP)SelectObject(hMemDC, hbm);

      if (hOldBitmap == NULL)
      return;   // destructors will clean up

      BitBlt(di.hdcDraw,
          di.prcBounds->left+size.cx + 2,
          di.prcBounds->top+1,
          size.cx,
          size.cy,
          hMemDC, 0,
          0,
          SRCCOPY);
```

```
        SelectObject(di.hdcDraw, hOldBitmap);
        DeleteDC(hMemDC);
}
```

```
void Cmyatldiceob::ShowSecondDieFace(ATL_DRAWINFO &di)
{
    BITMAP bmInfo;
    GetObject(m_dieBitmaps[m_nFirstDieValue-1],
       sizeof(bmInfo), &bmInfo);

    SIZE size;

    size.cx = bmInfo.bmWidth;
    size.cy = bmInfo.bmHeight;

    HDC hMemDC;
    hMemDC = CreateCompatibleDC(di.hdcDraw);

    HBITMAP hOldBitmap;
    HBITMAP hbm = m_dieBitmaps[m_nSecondDieValue-1];
    hOldBitmap = (HBITMAP)SelectObject(hMemDC, hbm);

    if (hOldBitmap == NULL)
      return;     // destructors will clean up

    BitBlt(di.hdcDraw,
          di.prcBounds->left+size.cx + 2,
          di.prcBounds->top+1,
          size.cx,
          size.cy,
          hMemDC, 0,
          0,
          SRCCOPY);

    SelectObject(di.hdcDraw, hOldBitmap);
    DeleteDC(hMemDC);
}
```

Listing 9.

Add the `RollDice()` method using ClassView to `Imyatldiceob` interface.

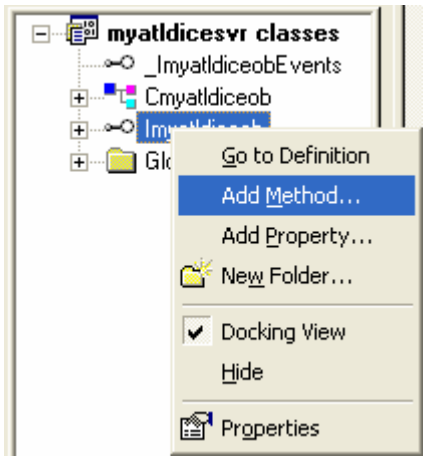

Figure 23: Adding method to `Imyatldiceob` interface.

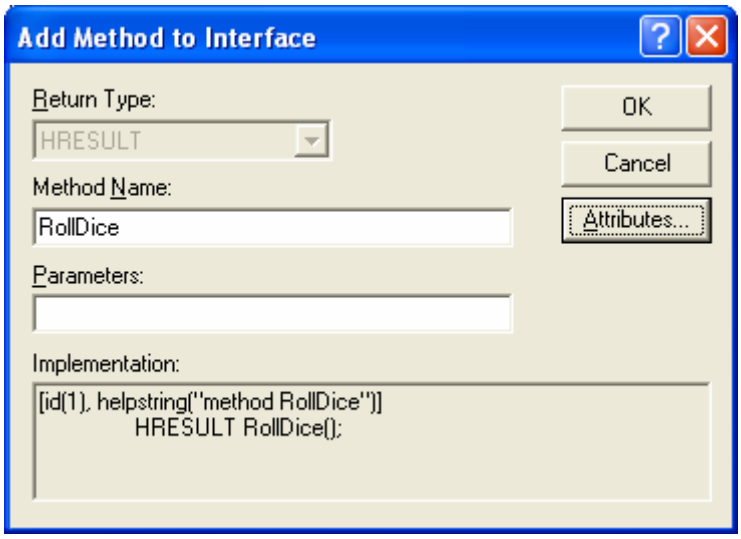

Figure 24: Adding `RollDice()` method to `Imyatldiceob` interface.

```
// Imyatldiceob
public:
    STDMETHOD(RollDice)();

    HRESULT OnDraw(ATL_DRA
    {
```

Listing 10.

Add the `RollDice()` code.

```
STDMETHODIMP Cmyatldiceob::RollDice()
{
      // TODO: Add your implementation code here
      SetTimer(1, 250);
      return S_OK;
}
```

```
STDMETHODIMP Cmyatldiceob::RollDice()
{
    // TODO: Add your implementation code here
    SetTimer(1, 250);
    return S_OK;
}
```

Listing 11.

Add `WM_LBUTTONDBLCLK` Windows message to `Cmyatldiceob` class using ClassView.

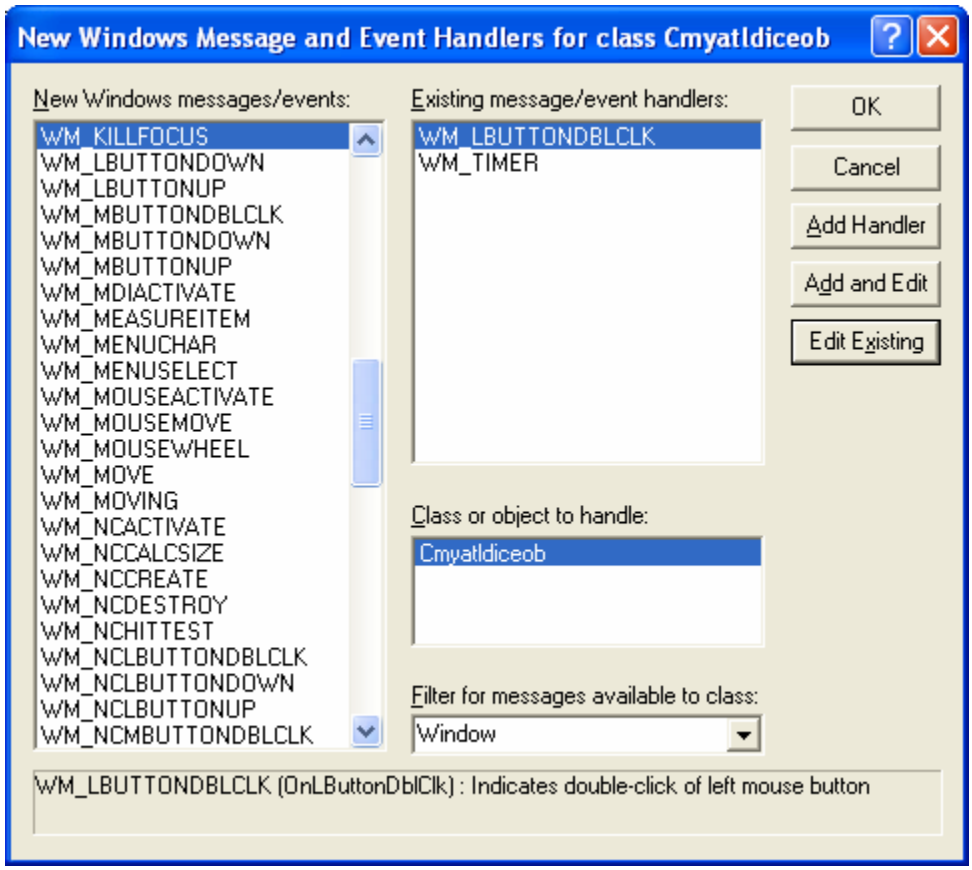

Figure 25: Add `WM_LBUTTONDBLCLK` Windows message to `Cmyatldiceob` class.

Then, add the code.

```
LRESULT OnLButtonDblClk(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
	// TODO : Add Code for message handler. Call DefWindowProc if
necessary.
	RollDice();
	bHandled = TRUE;
	return 0;
}
```

```
LRESULT OnLButtonDblClk(UINT uMsg, WPARAM wParam,
    LPARAM lParam, BOOL& bHandled)
{
    // TODO : Add Code for message handler.
    // Call DefWindowProc if necessary.
    RollDice();
    bHandled = TRUE;
    return 0;
}
```

Listing 12.

Build and run using container or use the **ActiveX Control Test Container** as shown below. Complete steps to build a dialog based program to test this dice control can be found HERE.

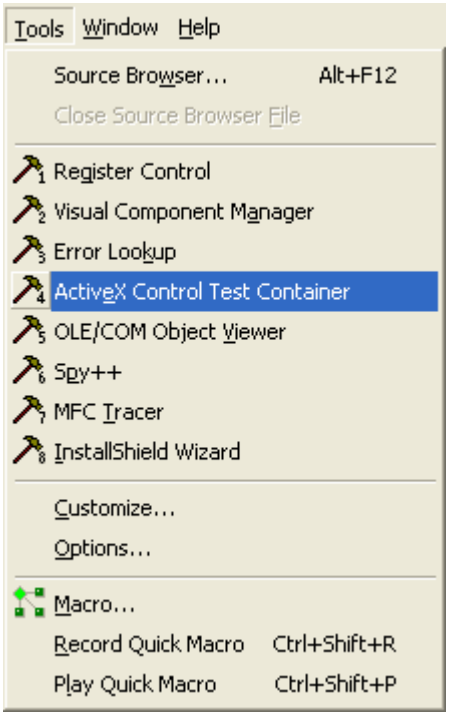

Figure 26: Using **ActiveX Control Test Container** to test our control.

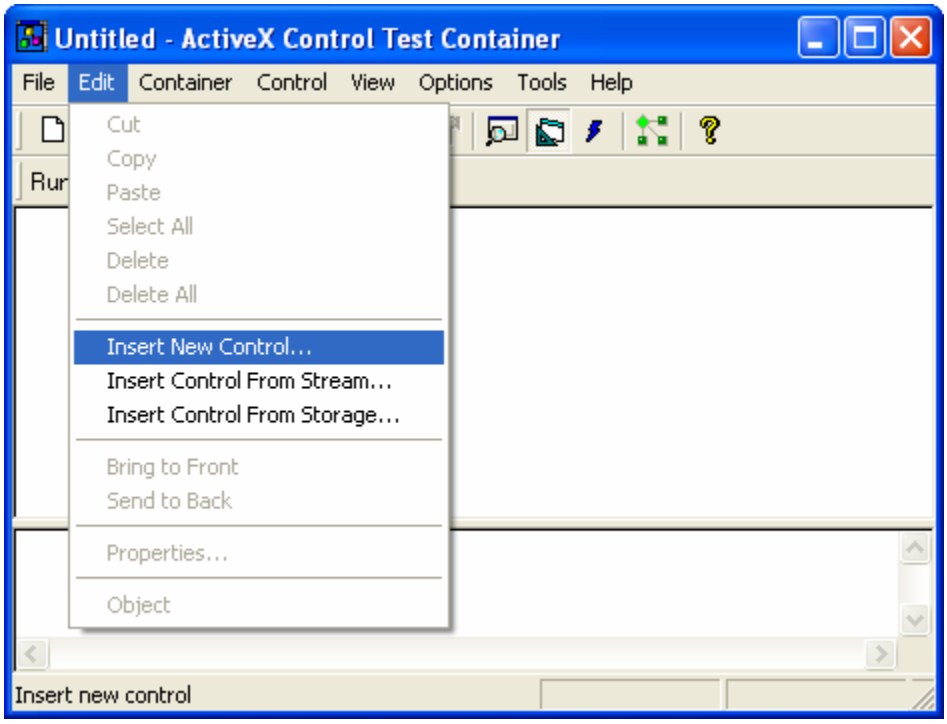

Figure 27: Inserting ActiveX control for testing.

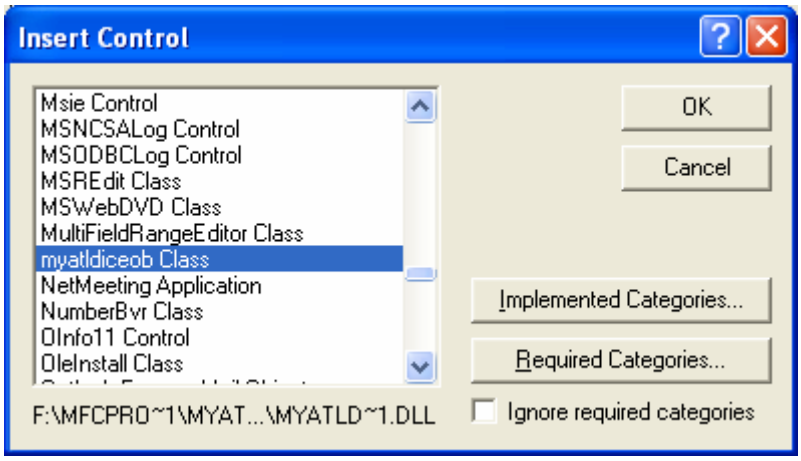

Figure 28: Selecting **myatldiceob** control for testing.

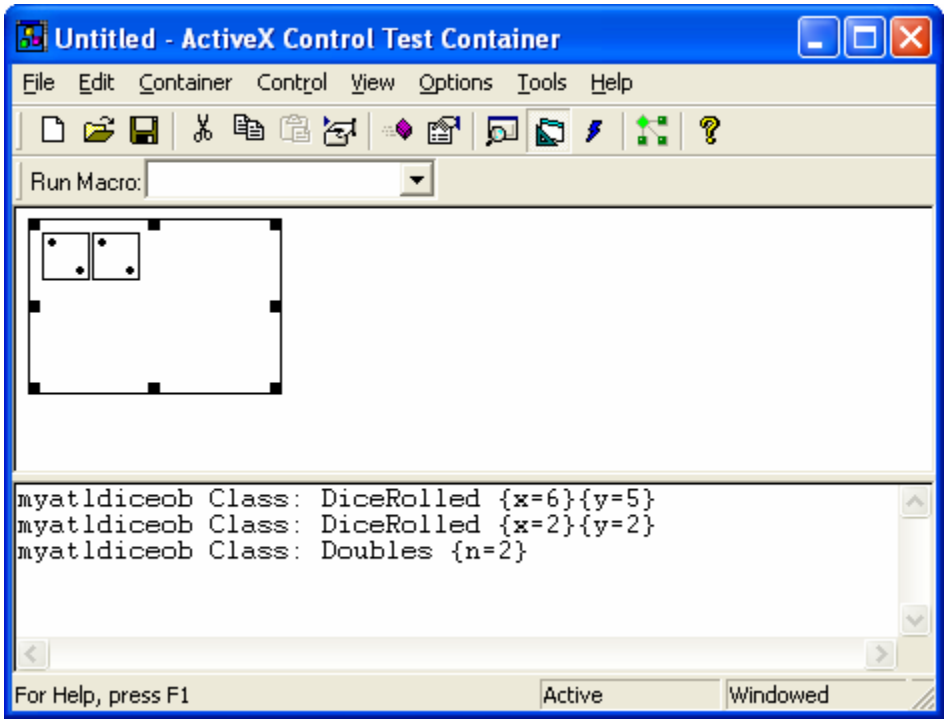

Figure 29: **myatldiceob** in ActiveX Control Test Container.

Add other functionalities. Add the following properties for get and put functions to `Imyatldiceob` interface.

```
DiceColor()
TimesToRoll()
```

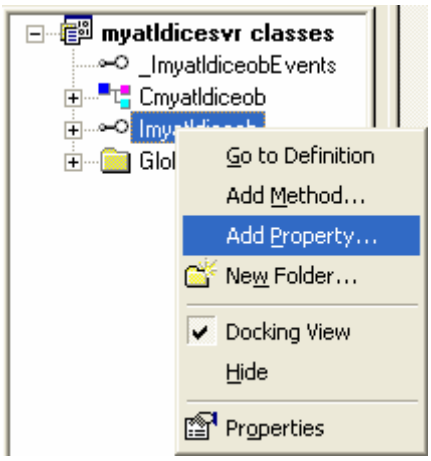

Figure 30: Adding properties for get and put functions to `Imyatldiceob` interface.

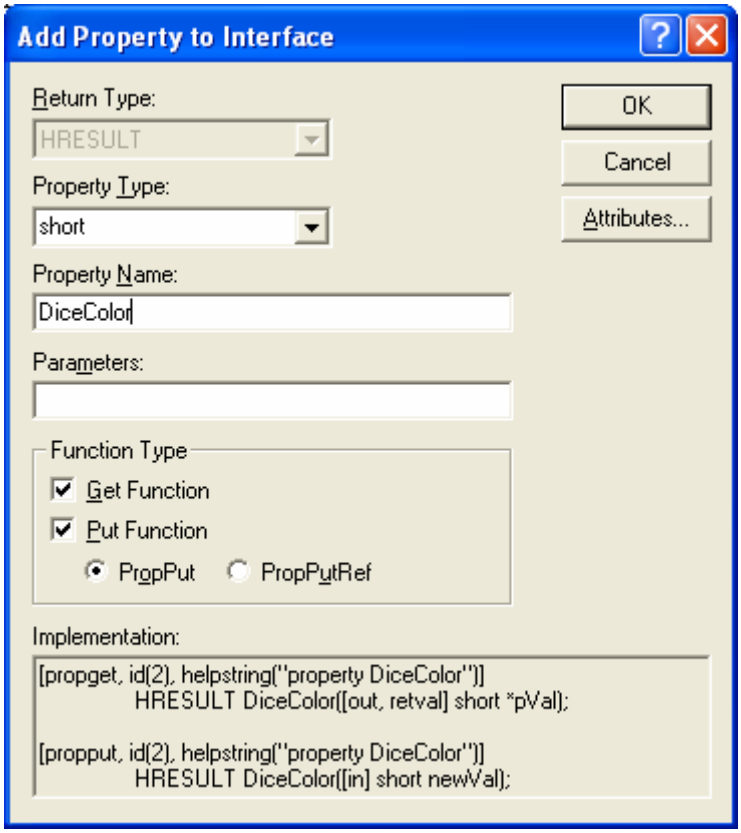

Figure 31: Adding `DiceColor()` property for get and put functions to `Imyatldiceob` interface.

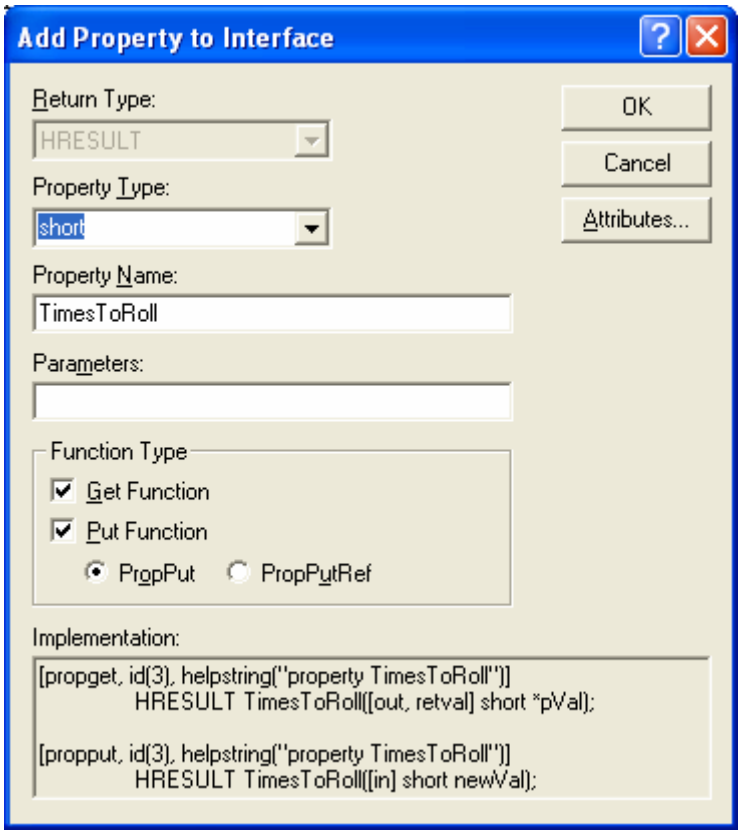

Figure 32: Adding `TimesToRoll()` property for get and put functions to `Imyatldiceob` interface.

You can verify the previous step in **myatldiceob.h** file as shown below.

```
// Imyatldiceob
public:
    STDMETHOD(get_TimesToRoll)(/*[out, retval]*/ short *pVal);
    STDMETHOD(put_TimesToRoll)(/*[in]*/ short newVal);
    STDMETHOD(get_DiceColor)(/*[out, retval]*/ short *pVal);
    STDMETHOD(put_DiceColor)(/*[in]*/ short newVal);
    STDMETHOD(RollDice)();
```

Listing 13.

And also in **myatldicesvr.idl** file.

```
    interface Imyatldiceob : IDispatch
    {
        [propput, id(DISPID_BACKCOLOR)]
        HRESULT BackColor([in]OLE_COLOR clr);
        [propget, id(DISPID_BACKCOLOR)]
        HRESULT BackColor([out,retval]OLE_COLOR* pclr);
        [id(1), helpstring("method RollDice")] HRESULT RollDice();
        [propget, id(2), helpstring("property DiceColor")]
         HRESULT DiceColor([out, retval] short *pVal);
        [propput, id(2), helpstring("property DiceColor")]
         HRESULT DiceColor([in] short newVal);
        [propget, id(3), helpstring("property TimesToRoll")]
         HRESULT TimesToRoll([out, retval] short *pVal);
        [propput, id(3), helpstring("property TimesToRoll")]
         HRESULT TimesToRoll([in] short newVal);
    };
```

Listing 14.

Add codes to the previously added handlers in **myatldiceob.cpp** file.

```
STDMETHODIMP Cmyatldiceob::get_DiceColor(short *pVal)
{
	// TODO: Add your implementation code here
	*pVal = m_nDiceColor;
	return S_OK;
}

STDMETHODIMP Cmyatldiceob::put_DiceColor(short newVal)
{
	// TODO: Add your implementation code here
	m_nDiceColor = newVal;
	LoadBitmaps();
	FireViewChange();
	return S_OK;
}

STDMETHODIMP Cmyatldiceob::get_TimesToRoll(short *pVal)
{
	// TODO: Add your implementation code here
	*pVal = m_nTimesToRoll;
	return S_OK;
}

STDMETHODIMP Cmyatldiceob::put_TimesToRoll(short newVal)
{
	// TODO: Add your implementation code here
	m_nTimesToRoll = newVal;
	return S_OK;
}
```

```
STDMETHODIMP Cmyatldiceob::get_DiceColor(short *pVal)
{
    // TODO: Add your implementation code here
    *pVal = m_nDiceColor;
    return S_OK;
}

STDMETHODIMP Cmyatldiceob::put_DiceColor(short newVal)
{
    // TODO: Add your implementation code here
    m_nDiceColor = newVal;
    LoadBitmaps();
    FireViewChange();
    return S_OK;
}

STDMETHODIMP Cmyatldiceob::get_TimesToRoll(short *pVal)
{
    // TODO: Add your implementation code here
    *pVal = m_nTimesToRoll;
    return S_OK;
}

STDMETHODIMP Cmyatldiceob::put_TimesToRoll(short newVal)
{
    // TODO: Add your implementation code here
    m_nTimesToRoll = newVal;
    return S_OK;
}
```

Listing 15.

Add methods to `_ImyatldiceobEvents` interface. These methods are visible in **myatldicesvr.idl** file only. Then, they will be visible to the `myatldiceob` class through the **Implement Connection Point**.

```
void Doubles(short n)
void SnakeEyes()
void DiceRolled(short x, short y)
```

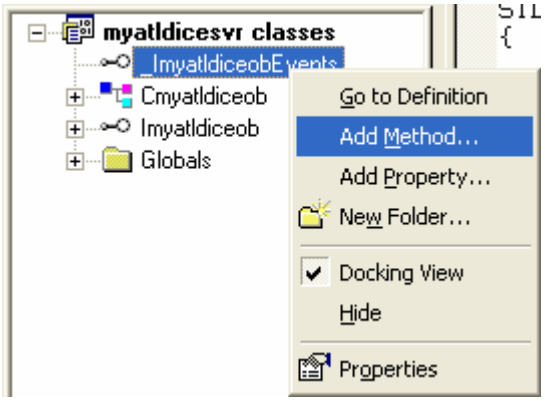

Figure 33: Adding methods to `_ImyatldiceobEvents` interface.

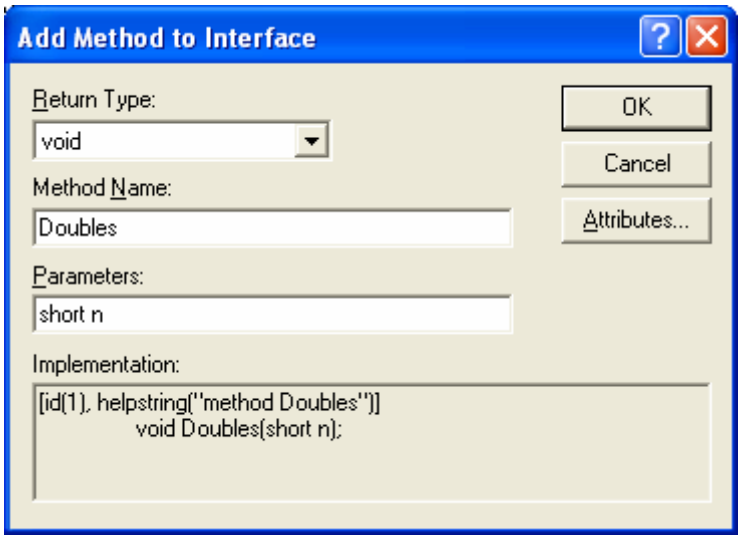

Figure 34: Adding `Doubles()` methods to `_ImyatldiceobEvents` interface.

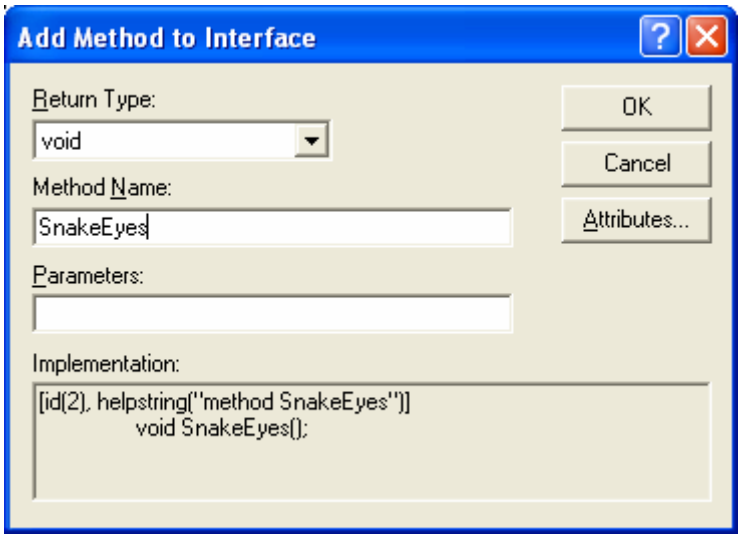

Figure 35: Adding `SnakeEyes()` methods to `_ImyatldiceobEvents` interface.

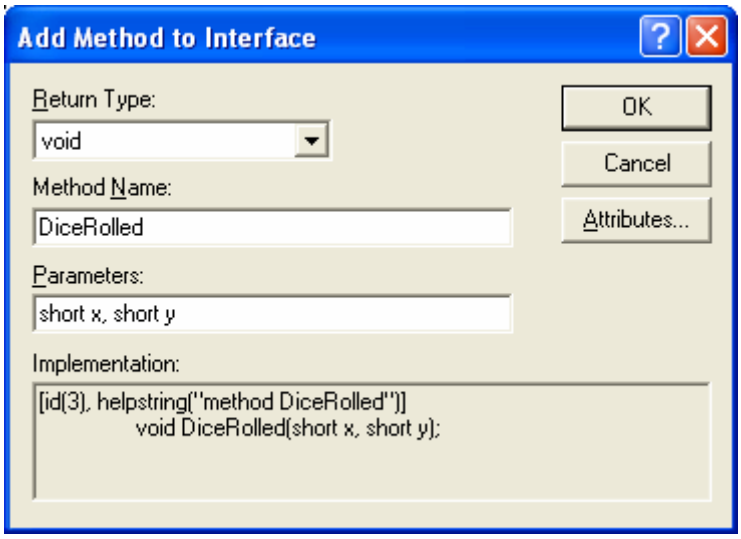

Figure 36: Adding `DiceRolled()` methods to `_ImyatldiceobEvents` interface.

Add the **Implement Connection Point** functionality to `Cmyatldiceob` class for the previously created methods.

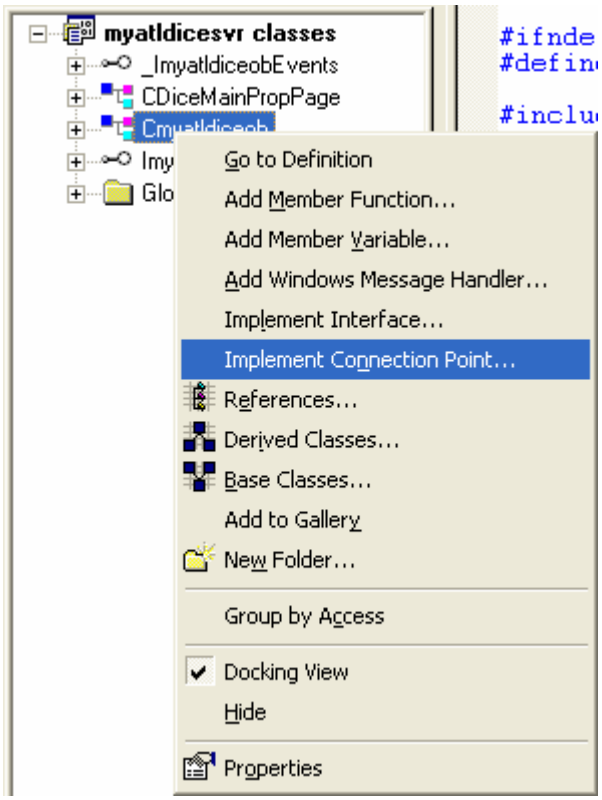

Figure 37: Adding **Implement Connection Point** functionality to `Cmyatldiceob` class.

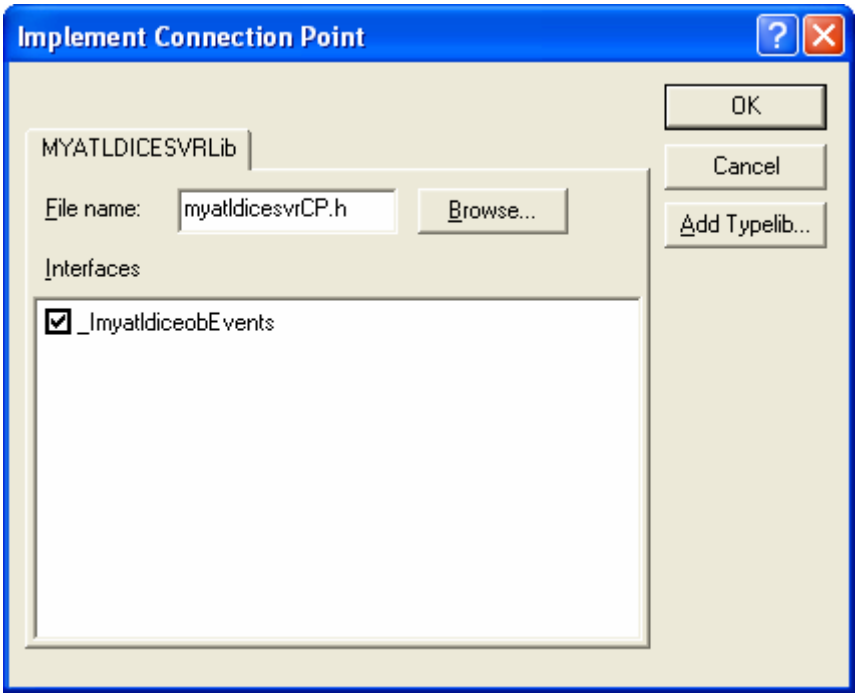

Figure 38: Selecting `_ImyatldiceobEvents` event interface.

Edit the `LoadBitmap()` function by adding the `switch` statement to select different dice color.

```
BOOL Cmyatldiceob::LoadBitmaps()
{
	int i;
	BOOL bSuccess = TRUE;
```

```
      int nID = IDB_DICE1;

      switch(m_nDiceColor)
      {
        case 0:
            nID = IDB_DICE1;
            break;

        case 1:
            nID = IDB_BLUEDICE1;
            break;

        case 2:
            nID = IDB_REDDICE1;
            break;
      }

   for(i=0; i<MAX_DIEFACES; i++)
   {
      DeleteObject(m_dieBitmaps[i]);
      m_dieBitmaps[i] = LoadBitmap(_Module.m_hInst, MAKEINTRESOURCE(nID+i));
      if(!m_dieBitmaps[i])
      {
          ::MessageBox(NULL,
                 "Failed to load bitmaps",
                  NULL,
                  MB_OK);
          bSuccess = FALSE;
      }
   }
   OutputDebugString("Got to the LoadBitmaps functions\n");
   return bSuccess;
}
```

```
// Cmyatldiceob

BOOL Cmyatldiceob::LoadBitmaps()
{
    int i;
    BOOL bSuccess = TRUE;
    int nID = IDB_DICE1;

    switch(m_nDiceColor)
    {
        case 0:
            nID = IDB_DICE1;
            break;

        case 1:
            nID = IDB_BLUEDICE1;
            break;

        case 2:
            nID = IDB_REDDICE1;
            break;
    }

   for(i=0; i<MAX_DIEFACES; i++)
```

Listing 16.

Edit the `RollDice()` function.

```
STDMETHODIMP Cmyatldiceob::RollDice()
```

```
{
	// TODO: Add your implementation code here
	if(::IsWindow(m_hWnd))
	{
	    SetTimer(1, 250);
	}
	return S_OK;
}
```

```
STDMETHODIMP Cmyatldiceob::RollDice()
{
    // TODO: Add your implementation code here
    if(::IsWindow(m_hWnd))
    {
        SetTimer(1, 250);
    }
    return S_OK;
}
```

Listing 17.

Edit `OnTimer()` function to reflect our new functionality.

```
LRESULT OnTimer(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
	// TODO : Add Code for message handler.
	// Call DefWindowProc if necessary.
	if(m_nTimesRolled > m_nTimesToRoll)
	{
		m_nTimesRolled = 0;
		KillTimer(1);

		Fire_DiceRolled(m_nFirstDieValue, m_nSecondDieValue);

		if(m_nFirstDieValue == m_nSecondDieValue)
		{
			Fire_Doubles(m_nFirstDieValue);
		}

	if(m_nFirstDieValue == 1 && m_nSecondDieValue == 1)
	{
		Fire_SnakeEyes();
	}
	}
	else
	{
		m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
		m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
		FireViewChange();
		m_nTimesRolled++;
	}

	bHandled = TRUE;
	return 0;
}
```

Adding a property page so that user can easily use/change our control properties. Add new ATL object by selecting the **Insert New ATL Object**.

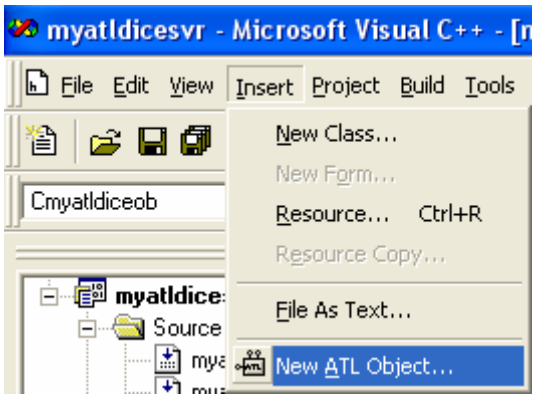

Figure 39: Inserting new ATL object.

Select **Controls** in **Category** list and **Property Page** in **Objects** list. Then click the **Next** button

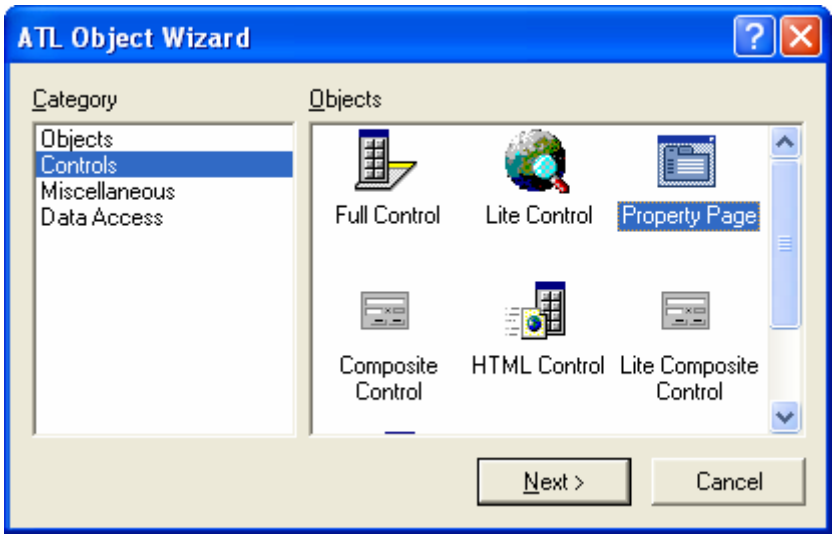

Figure 40: Selecting **Property Object** for control's property page.

Type in a meaningful name such as `DiceMainPropPage` as shown below in the **Short Name** field.

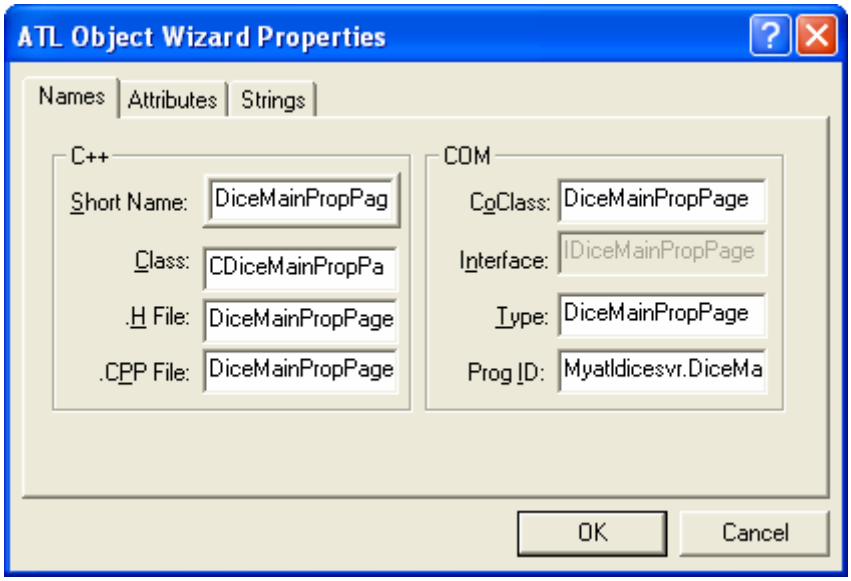

Figure 41: Entering new ATL object's name.

Just accept the default setting for **Attributes** page.

Figure 42: Accepting the defaults option of the **Attributes**.

Type in meaningful strings for **Strings** page and click the **OK** button.

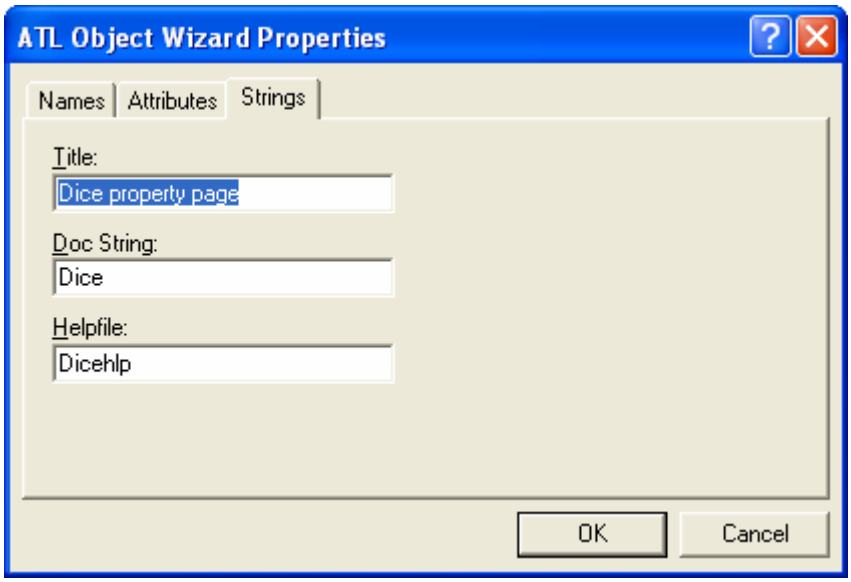

Figure 43: Entering some strings for our control's property page.

Add **Static text**, **Combo box** and **Edit control** to the blank `IDD_DICEMAINPROPPAGE` template. Use `IDC_COLOR` and `IDC_TIMESTOROLL` as their `ID`s respectively. Leave others as default.

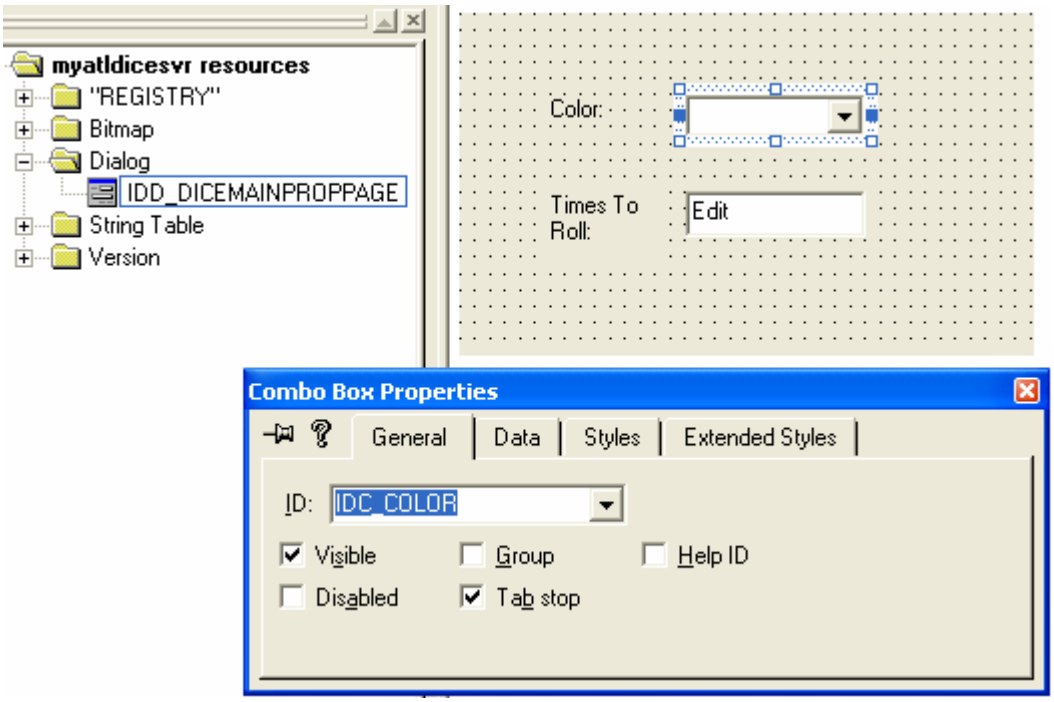

Figure 44: Adding Static text, Combo box and Edit control to the blank property page template.

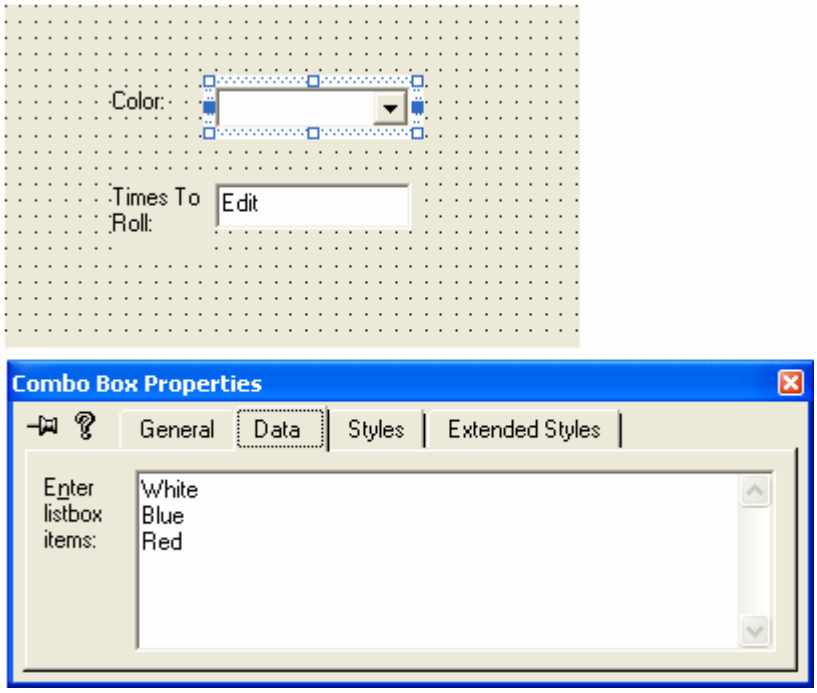

Figure 45: Combo box, `IDC_COLOR` property.

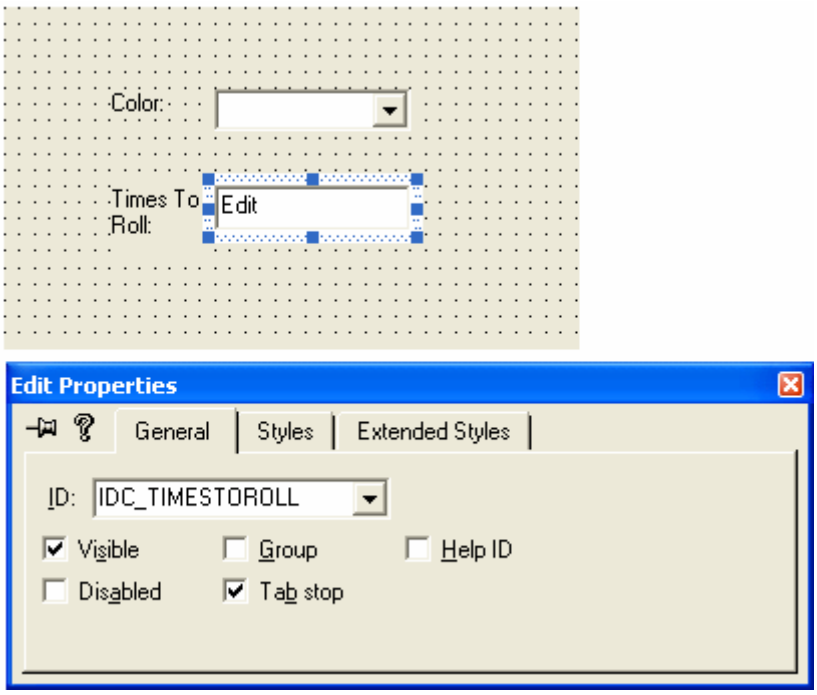

Figure 46: Edit control's property.

Add Windows message handlers as shown in the following Table using ClassView to `CDiceMainPropPage` class.

ID/Class	Windows message	Functions
`IDC_COLOR`	`CBN_SELENDOK`	`OnColorChange()`
`IDC_TIMESTOROLL`	`EN_CHANGE`	`OnTimesToRollChange()`
`CDiceMainPropPage`	`WM_INITDIALOG`	-

Table 1.

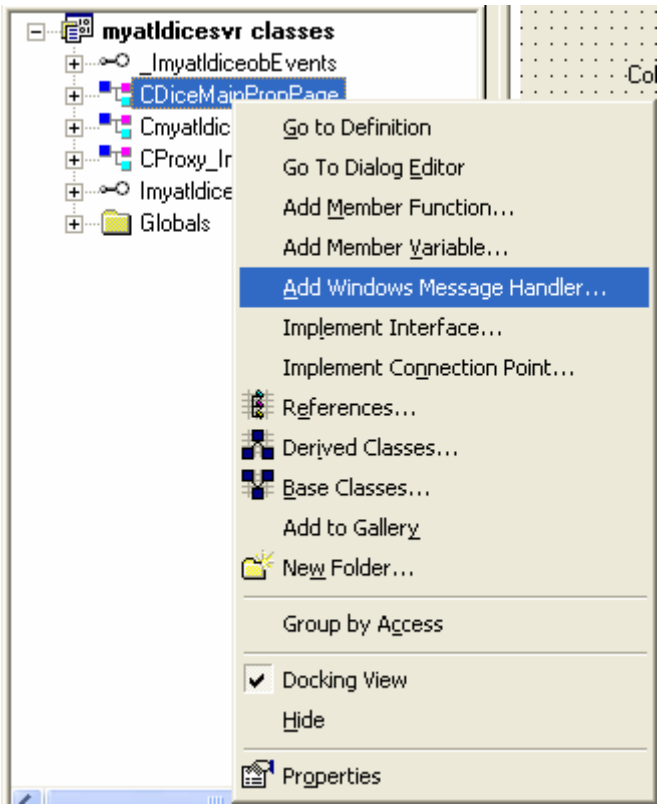

Figure 47: Adding Windows message handlers to `CDiceMainPropPage` class.

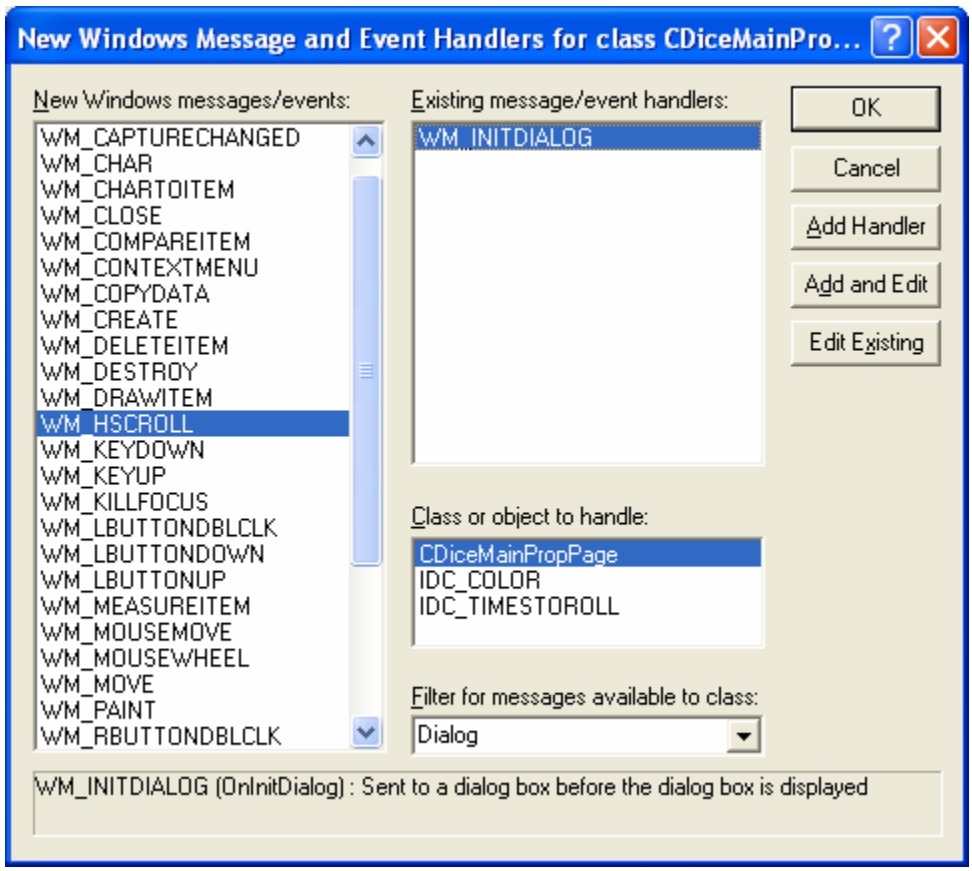

Figure 48: Adding `WM_INITDIALOG` handler to `CDiceMainPropPage` class.

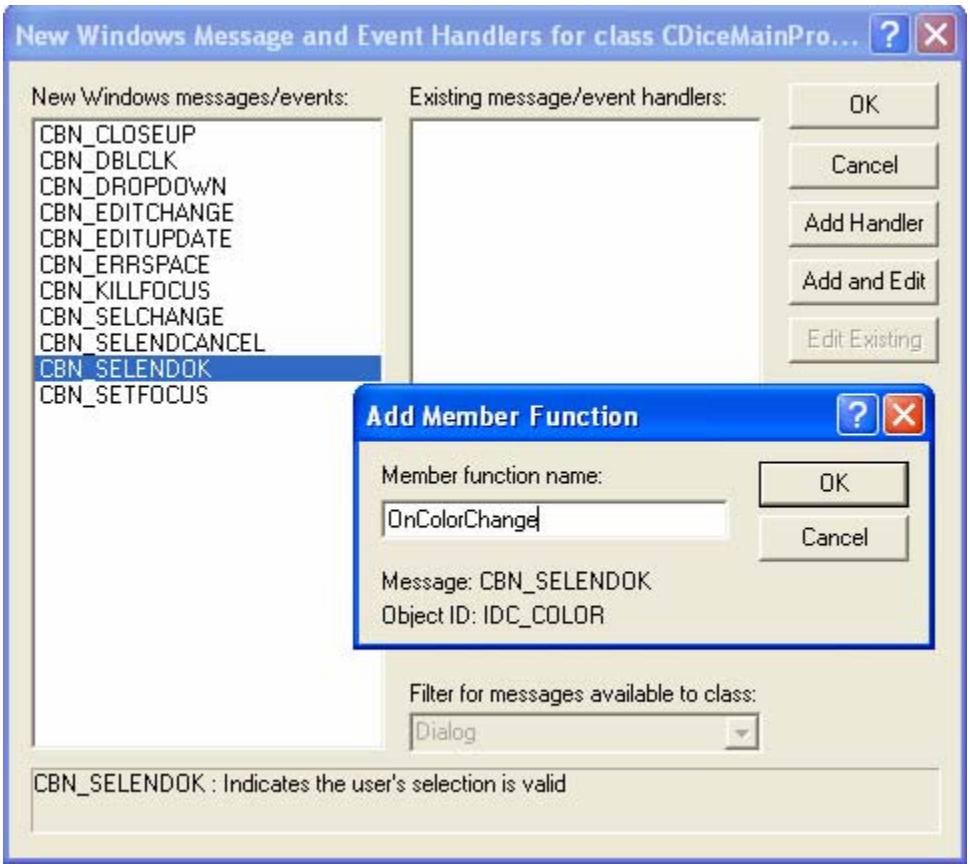

Figure 49: Adding `CBN_SELENDOK` handler to `CDiceMainPropPage` class.

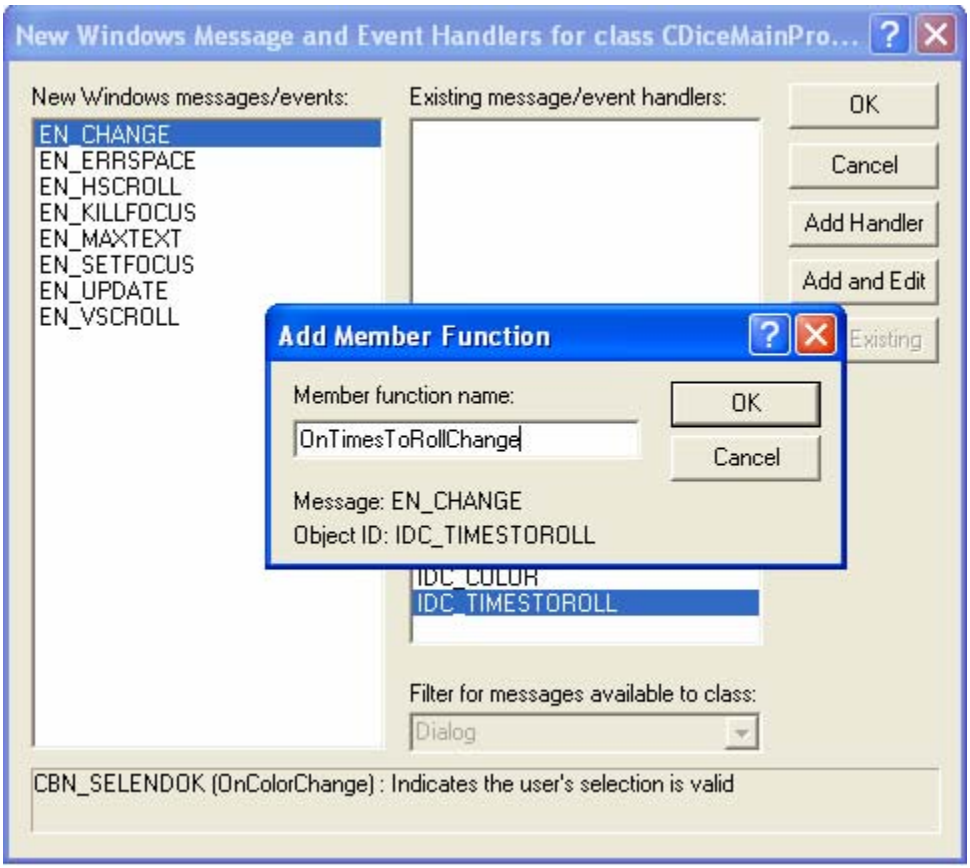

Figure 50: Adding `EN_CHANGE` handler to `CDiceMainPropPage` class.

Before we forget, add the following property entries to **myatldiceob.h** file manually.

```
PROP_ENTRY("DiceColor", 2, CLSID_DiceMainPropPage)
PROP_ENTRY("TimesToRoll", 3, CLSID_DiceMainPropPage)
```

```
BEGIN_PROP_MAP(Cmyatldiceob)
    PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
    PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
    PROP_ENTRY("BackColor", DISPID_BACKCOLOR, CLSID_StockColorPage)
    PROP_ENTRY("DiceColor", 2, CLSID_DiceMainPropPage)
    PROP_ENTRY("TimesToRoll", 3, CLSID_DiceMainPropPage)
    // Example entries
    // PROP_ENTRY("Property Description", dispid, clsid)
    // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()
```

Listing 18.

Edit the `Apply()` method in **DiceMainPropPage.h** file.

```
STDMETHOD(Apply)(void)
{
      ATLTRACE(_T("CDiceMainPropPage::Apply\n"));
      for (UINT i = 0; i < m_nObjects; i++)
      {
            USES_CONVERSION;
            ATLTRACE(_T("CDiceMainPropPage::Apply\n"));
            for (UINT i = 0; i < m_nObjects; i++)
            {
            CComQIPtr<IATLDiceObj, &IID_IATLDiceObj> pATLDiceObj(m_ppUnk[i]);
            HWND hWndComboBox = GetDlgItem(IDC_COLOR);
            short nColor = (short)::SendMessage(hWndComboBox,
                                                CB_GETCURSEL,
                                                0, 0);
            if(nColor >= 0 && nColor <= 2) {
            if FAILED(pATLDiceObj->put_DiceColor(nColor))
            {
                  CComPtr<IErrorInfo> pError;
                  CComBSTR            strError;
                  GetErrorInfo(0, &pError);
                  pError->GetDescription(&strError);
                  MessageBox(OLE2T(strError),
                              _T("Error"),
                              MB_ICONEXCLAMATION);
                  return E_FAIL;
            }
            }
            short nTimesToRoll = (short)GetDlgItemInt(IDC_TIMESTOROLL);
            if FAILED(pATLDiceObj->put_TimesToRoll(nTimesToRoll))
            {
                  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;

            }
            m_bDirty = FALSE;
            return S_OK;
      }
```

Add the handlers' codes.

```
LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL& bHandled)
{
      // TODO : Add Code for message handler. Call DefWindowProc if
necessary.
      HWND hWndComboBox = GetDlgItem(IDC_COLOR);
            ::SendMessage(hWndComboBox,
                        CB_ADDSTRING,
                        0, (LPARAM)"White");
            ::SendMessage(hWndComboBox,
                        CB_ADDSTRING,
                        0, (LPARAM)"Blue");
            ::SendMessage(hWndComboBox,
                        CB_ADDSTRING,
                        0, (LPARAM)"Red");
      bHandled = TRUE;
      return 0;
}


LRESULT OnColorChange(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
      // TODO : Add Code for control notification handler.
      SetDirty(TRUE);
      return 0;
}

LRESULT OnTimesToRollChange(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled)
{
      // TODO : Add Code for control notification handler.
      SetDirty(TRUE);
      return 0;
}
```

Manually add the `Show()` and `SetObjects()` methods. Take note that if you use different interface name for the ATL object name for example, you have to change the related codes accordingly.

```
STDMETHOD(Show)(UINT nCmdShow)
{
      HRESULT hr;
      USES_CONVERSION;

      if(nCmdShow == SW_SHOW || nCmdShow == SW_SHOWNORMAL)
      {
      for (UINT i = 0; i < m_nObjects; i++)
      {
        CComQIPtr<Imyatldiceob, &IID_Imyatldiceob> pmyatldiceob(m_ppUnk[i]);
            short nColor = 0;

            if FAILED(pmyatldiceob->get_DiceColor(&nColor))
            {
```

```
                CComPtr<IErrorInfo> pError;
                CComBSTR            strError;
                GetErrorInfo(0, &pError);
                pError->GetDescription(&strError);
                MessageBox(OLE2T(strError), _T("Error"),
                MB_ICONEXCLAMATION);
                return E_FAIL;
            }
            HWND hWndComboBox = GetDlgItem(IDC_COLOR);
            ::SendMessage(hWndComboBox,
                        CB_SETCURSEL,
                        nColor, 0);

            short nTimesToRoll = 0;
            if FAILED(pmyatldiceob->get_TimesToRoll(&nTimesToRoll))
            {
                CComPtr<IErrorInfo> pError;
                CComBSTR            strError;
                GetErrorInfo(0, &pError);
                pError->GetDescription(&strError);
                MessageBox(OLE2T(strError), _T("Error"),
                MB_ICONEXCLAMATION);
                return E_FAIL;
            }
                SetDlgItemInt(IDC_TIMESTOROLL, nTimesToRoll, FALSE);
            }
      }
      m_bDirty = FALSE;

      hr = IPropertyPageImpl<CDiceMainPropPage>::Show(nCmdShow);
    return hr;
}

STDMETHOD(SetObjects)(ULONG nObjects, IUnknown** ppUnk)
{
      HRESULT hr = IPropertyPageImpl<CDiceMainPropPage>::SetObjects(nObjects,
ppUnk);
      return hr;
}
```

Build and make sure there is no error. Try using the dice control in Visual C++ dialog based program or Visual Basic form. The following shows the dice control using the **ActiveX Control Test Container**. Double click the dice or in the square area. Here, we cannot see/use the property page.

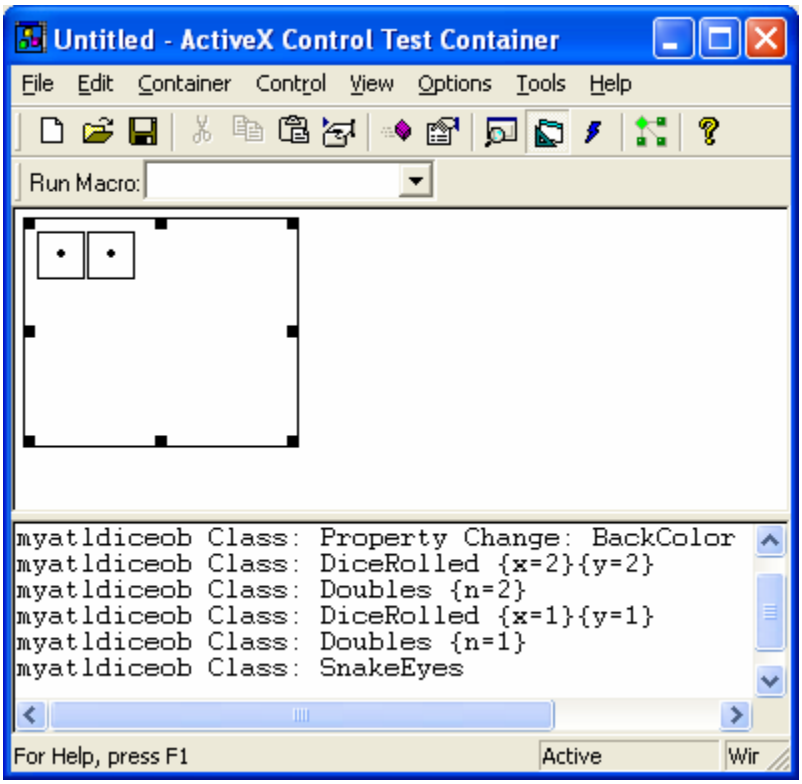

Figure 51: **myatldicesvr** in ActiveX Control Test Container.

The Story

As always, the easiest way to create a COM server in ATL is to use the **ATL COM Object Wizard**. To use the ATL COM Object Wizard, select **New** from the **File** menu. Select the **Project** tab in the **New** dialog, and highlight the **ATL COM AppWizard** item. Name the project something clever like **myatldicesvr**. As you step through AppWizard, just leave the defaults checked. Doing so will ensure that the server you create is a DLL. Once the DLL server has been created, perform the following steps:

Select **New ATL Object** from the **Insert** menu to insert a new ATL object into the project.

In the **ATL Object Wizard**, select **Controls** from the **Category** list and then select **Full Control** from the **Objects** list.

Click **Next** to open the **ATL Object Wizard Properties** dialog. In the **Short Name** text box on the **Names** tab, give the control some clever name (like **myatldiceob**). The dialog box should look similar to Figure 52.

Figure 52: The ATL Object Wizard Properties dialog box.

Select the **Attributes** tab. Here's where you configure the control. For example, you can:

1. Designate the threading model for the control.
2. Decide whether the main interface is a dual or custom interface.
3. Indicate whether your control supports aggregation.
4. Choose whether you want to use COM exceptions and connection points in your control.

To make your life easier for now, select **Support Connection Points**. This will save you some typing later on. Leave everything else as the default value. Figure 53 shows what the **Attributes** tab on the ATL Object Wizard Properties dialog box looks like now.

Select the **Miscellaneous** tab. Here you have the option of applying some miscellaneous traits to your control. For example, you can give the control behaviors based on regular Microsoft Windows controls such as buttons and edit controls. You might also select other options for your control, such as having your control appear invisible at runtime or giving your control an opaque background. Figure 54 shows the available options.

Figure 53: The Attributes tab on the ATL Object Wizard Properties dialog box.

Figure 54: The Miscellaneous control properties tab on the ATL Object Wizard Properties dialog box.

Finally, select the **Stock Properties** tab if you want to give your control some stock properties. Stock properties are those properties that you might expect any control to have, including background colors, border colors, foreground colors, and a caption. Figure 55 shows the Stock Properties tab.

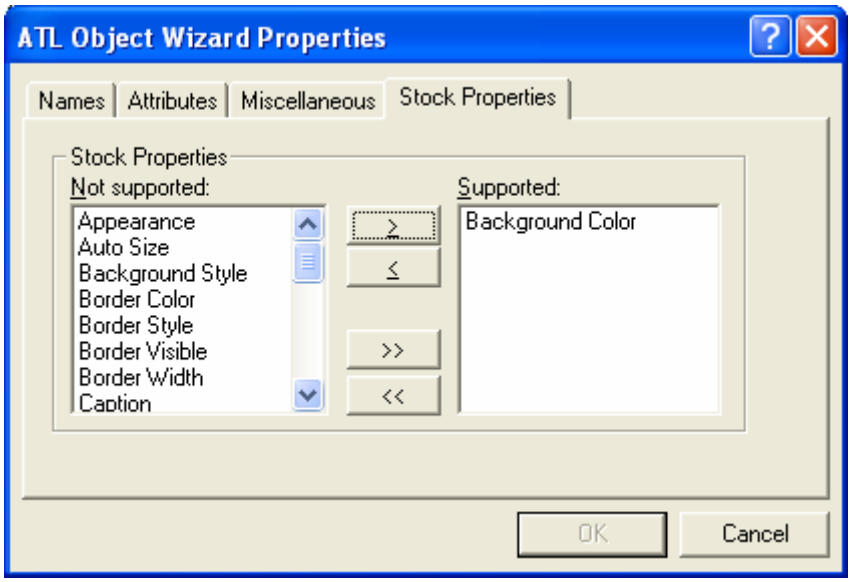

Figure 55: The Stock Properties tab on the ATL Object Wizard Properties dialog box.

When you've finished selecting the attributes for the control, click **OK**.

The ATL Object Wizard adds a header file and a source file defining the new control. In addition, the ATL Object Wizard sets aside space in the IDL file to hold the control's main interface and assigns a `GUID` to the interface. Here's the C++ definition of the control produced by the ATL Object Wizard (**myatldiceob.h**):

```
// myatldiceob.h : Declaration of the Cmyatldiceob

#ifndef __MYATLDICEOB_H_
#define __MYATLDICEOB_H_

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

```
/////////////////////////////////////////////////////////////////////////////
// Cmyatldiceob
class ATL_NO_VTABLE Cmyatldiceob :
	public CComObjectRootEx<CComSingleThreadModel>,
	public IDispatchImpl<Imyatldiceob, &IID_Imyatldiceob,
&LIBID_MYATLDICESVRLib>,
	public CComControl<Cmyatldiceob>,
	public IPersistStreamInitImpl<Cmyatldiceob>,
	public IOleControlImpl<Cmyatldiceob>,
	public IOleObjectImpl<Cmyatldiceob>,
	public IOleInPlaceActiveObjectImpl<Cmyatldiceob>,
	public IViewObjectExImpl<Cmyatldiceob>,
	public IOleInPlaceObjectWindowlessImpl<Cmyatldiceob>,
	public IConnectionPointContainerImpl<Cmyatldiceob>,
	public IPersistStorageImpl<Cmyatldiceob>,
	public ISpecifyPropertyPagesImpl<Cmyatldiceob>,
	public IQuickActivateImpl<Cmyatldiceob>,
	public IDataObjectImpl<Cmyatldiceob>,
	public IProvideClassInfo2Impl<&CLSID_myatldiceob,
&DIID__ImyatldiceobEvents, &LIBID_MYATLDICESVRLib>,
	public IPropertyNotifySinkCP<Cmyatldiceob>,
	public CComCoClass<Cmyatldiceob, &CLSID_myatldiceob>
{
   ...
   ...
   ...
};

#endif //__MYATLDICEOB_H_
```

That's a pretty long inheritance list. You've already seen the template implementations of `IUnknown` and support for class objects. They exist in `CComObjectRootEx` and `CComCoClass`. You've also seen how ATL implements `IDispatch` within the `IDispatchImpl` template. However, for a basic control there are about 11 more interfaces required to make everything work. These interfaces can be categorized into several areas as shown in the following table.

Category	Interface
Interfaces for handling self-description	`IProvideClassInfo2`
Interfaces for handling persistence	`IPersistStreamInit` `IPersistStorage`
Interfaces for handling activation	`IQuickActivate` (and some of `IOleObject`)
Interfaces from the original OLE Control specification	`IOleControl`
Interfaces from the OLE Document specification	`IOleObject`

Interfaces for rendering	`IOleInPlaceActiveObject`
	`IViewObject`
	`IOleInPlaceObjectWindowless`
	`IDataObject`

Interfaces for helping the container manage property pages	`ISpecifyPropertyPages`
Interfaces for handling connections	`IPropertyNotifySinkCP` `IConnectionPointContainer`

Table 2.

These are by and large boilerplate interfaces, ones that a COM class must implement to qualify as an ActiveX control. Most of the implementations are standard and vary only slightly (if at all) from one control to the next. The beauty of ATL is that it implements this standard behavior and gives you programmatic hooks where you can plug in your custom code. That way, you don't have to burn your eyes out by looking directly at the COM code. You can

live a full and rich life without understanding exactly how these interfaces work. However, if you want to know more about the internal workings of ActiveX Controls, be sure to check out these books: Inside OLE by Kraig Brockschmidt (Microsoft Press, 1995), ActiveX Controls Inside Out by Adam Denning (Microsoft Press, 1997), and Designing and Using ActiveX Controls by Tom Armstrong (IDG Books Worldwide, 1997).

ATL's Control Architecture

From the highest level, an ActiveX control has two aspects to it: its external state (what it renders on the screen) and its internal state (its properties). Once an ActiveX control is hosted by some sort of container (such as a Microsoft Visual Basic form or an MFC dialog box), it maintains a symbiotic relationship with that container. The client code talks to the control through incoming COM interfaces such as `IDispatch` and **OLE Document** interfaces like `IOleObject` and `IDataObject`.
The control also has the opportunity to talk back to the client. One method of implementing this two-way communication is for the client to implement an `IDispatch` interface to represent the control's event set. The container maintains a set of properties called **ambient properties** that the control can use to find out about its host. For instance, a control can camouflage itself within the container because the container makes the information stored in these properties available through a specifically named `IDispatch` interface. The container can implement an interface named `IPropertyNotifySink` to find out when the properties within a control might change. Finally, the container implements `IOleClientSite` and `IOleControlSite` as part of the control-embedding protocol.
The interfaces listed allow the client and the object to exhibit the behaviors expected of an ActiveX control. We'll tackle some of these interfaces as we go along. The best place to begin looking at ATL-based controls is the CComControl class and its base classes.

`CComControl`

You can find the definition of `CComControl` in Microsoft's **`ATLCTL.H`** file under ATL's **Include** directory. `CComControl` is a template class that takes a single class parameter:

```
template <class T>
class ATL_NO_VTABLE CComControl :  public CComControlBase,
                                   public CWindowImpl<T>
{
  ...
  ...
};
```

`CComControl` is a rather lightweight class that does little by itself; it derives functionality from `CComControlBase` and `CWindowImpl`. `CComControl` expects the template parameter to be an ATL-based COM object derived from `CComObjectRootEx`. `CComControl` requires the template parameter for various reasons, the primary reason being that from time to time the control class uses the template parameter to call back to the control's `InternalQueryInterface()`.
`CComControl` implements several functions that make it easy for the control to call back to the client. For example, `CComControl` implements a function named `FireOnRequestEdit()` to give controls the ability to tell the client that a specified property is about to change. This function calls back to the client through the client-implemented interface `IPropertyNotifySink`. `FireOnRequestEdit()` notifies all connected `IPropertyNotifySink` interfaces that the property specified by a certain `DISPID` is about to change. `CComControl` also implements the `FireOnChanged()` function. `FireOnChanged()` is very much like `FireOnRequestEdit()` in that it calls back to the client through the `IPropertyNotifySink` interface. This function tells the clients of the control (all clients connected to the control through `IPropertyNotifySink`) that a property specified by a certain `DISPID` has already changed.
In addition to mapping the `IPropertyNotifySink` interface to some more easily understood functions, `CComControl` implements a function named `ControlQueryInterface()`, which simply forwards on to the control's `IUnknown` interface. This is how you can get a control's IUnknown interface from inside the control. Finally, `CComControl` implements a function named `CreateControlWindow()`. The default behavior for this function is to call `CWindowImpl::Create`. Notice that CComControl also derives from `CWindowImpl`. If you want to, you can override this function to do something other than create a single window. For example, you might want to create multiple windows for your control.

Most of the real functionality for `CComControl` exists within those two other classes, `CComControlBase` and `CWindowImpl`. Let's take a look at those classes now.

`CComControlBase`

`CComControlBase` is a much more substantial class than `CComControl`. To begin with, `CComControlBase` maintains all the pointers used by the control to talk back to the client. `CComControlBase` uses ATL's `CComPtr` smart pointer to include member variables that wrap the following interfaces implemented for calling back to the client:

- A wrapper for `IOleInPlaceSite`(`m_spInPlaceSite`).
- An advise holder for the client's data advise sink (`m_spDataAdviseHolder`).
- An OLE advise holder for the client's OLE advise sink (`m_spOleAdviseHolder`).
- A wrapper for `IOleClientSite` (`m_spClientSite`).
- A wrapper for `IAdviseSink` (`m_spAdviseSink`).

`CComControlBase` also uses ATL's `CComDispatchDriver` to wrap the client's dispatch interface for exposing its ambient properties.
`CComControlBase` is also where you'll find the member variables that contain the control's sizing and positioning information: `m_sizeNatural`, `m_sizeExtent`, and `m_rcPos`. The other important data member within `CComControlBase` is the control's window handle. Most ActiveX controls are UI gadgets and as such maintain a window. `CWindowImpl` and `CWindowImplBase` handle the windowing aspects of an ATL-based ActiveX control.

`CWindowImpl` and `CWindowImplBase`

`CWindowImpl` derives from `CWindowImplBase`, which in turn derives from `CWindow` and `CMessageMap`. As a template class, `CWindowImpl` takes a single parameter upon instantiation. The template parameter is the control being created. `CWindowImpl` needs the control type because `CWindowImpl` calls back to the control during window creation. Let's take a closer look at how ATL handles windowing.

ATL Windowing

Just as `CComControl` is relatively lightweight (most work happens in `CComControlBase`), `CWindowImpl` is also relatively lightweight. `CWindowImpl` more or less handles only window creation. In fact, that's the only function explicitly defined by `CWindowImpl`. `CWindowImpl::Create` creates a new window based on the window class information managed by a class named `_ATLWNDCLASSINFO`. There's an ASCII character version and a wide character version.

```
struct _ATL_WNDCLASSINFOA
{
    WNDCLASSEXA m_wc;
    LPCSTR m_lpszOrigName;
    WNDPROC pWndProc;
    LPCSTR m_lpszCursorID;
    BOOL m_bSystemCursor;
    ATOM m_atom;
    CHAR m_szAutoName[13];
    ATOM Register(WNDPROC* p)
    {
        return AtlModuleRegisterWndClassInfoA(&_Module, this, p);
    }
};

struct _ATL_WNDCLASSINFOW
{
    WNDCLASSEXW m_wc;
    LPCWSTR m_lpszOrigName;
    WNDPROC pWndProc;
```

```
    LPCWSTR m_lpszCursorID;
    BOOL m_bSystemCursor;
    ATOM m_atom;
    WCHAR m_szAutoName[13];
    ATOM Register(WNDPROC* p)
    {
        return AtlModuleRegisterWndClassInfoW(&_Module, this, p);
    }
};
```

Then ATL uses typedefs to alias this structure to a single class named `CWndClassInfo`:

```
typedef _ATL_WNDCLASSINFOA CWndClassInfoA;
typedef _ATL_WNDCLASSINFOW CWndClassInfoW;
#ifdef UNICODE
#define CWndClassInfo CWndClassInfoW
#else
#define CWndClassInfo CWndClassInfoA
#endif
```

`CWindowImpl` uses a macro named `DECLARE_WND_CLASS` to add window class information to a `CWindowImpl`-derived class. `DECLARE_WND_CLASS` also adds a function named `GetWndClassInfo()`. Here's the `DECLARE_WND_CLASS` macro:

```
#define DECLARE_WND_CLASS(WndClassName) \
static CWndClassInfo& GetWndClassInfo() \
{ \
    static CWndClassInfo wc = \
    { \
        { sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,\
          StartWindowProc, \
          0, 0, NULL, NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1), \
          NULL, WndClassName, NULL }, \
          NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
    }; \
    return wc; \
}
```

This macro expands to provide a `CWndClassInfo` structure for the control class. Because `CWndClassInfo` manages the information for a single window class, each window created through a specific instance of `CWindowImpl` will be based on the same window class.
`CWindowImpl` derives from `CWindowImplBaseT`. `CWindowImplBaseT` derives from `CWindowImplRoot`, which is specialized around the `CWindow` class and the `CControlWinTraits` classes like this:

```
template <class TBase = CWindow, class TWinTraits = CControlWinTraits>
class ATL_NO_VTABLE CWindowImplBaseT : public CWindowImplRoot< TBase >
{
 public:
   ...
   ...
   ...
};
```

`CWindowImplRoot` derives from `CWindow` (by default) and `CMessageMap`. `CWindowImplBaseT` manages the window procedure of a `CWindowImpl`-derived class. `CWindow` is a lightweight class that wraps window handles in the same way (but not as extensively) as MFC's `CWnd` class. `CMessageMap` is a tiny class that defines a single pure virtual function named `ProcessWindowMessage()`. ATL-based message-mapping machinery assumes this function is available, so ATL-based classes that want to use message maps need to derive from `CMessageMap`. Let's take a quick look at ATL message maps.

ATL Message Maps

The root of ATL's message mapping machinery lies within the `CMessageMap` class. ATL-based controls expose message maps by virtue of indirectly deriving from `CWindowImplBase`. In MFC, by contrast, deriving from `CCmdTarget` enables message mapping. However, just as in MFC, it's not enough to derive from a class that supports message maps. The message maps actually have to be there, and those message maps are implemented via macros.
To implement a message map in an ATL-based control, use message map macros. First ATL's `BEGIN_MSG_MAP` macro goes into the control class's header file. `BEGIN_MSG_MAP` marks the beginning of the default message map. `CWindowImpl::WindowProc` uses this default message map to process messages sent to the window. The message map directs messages either to the appropriate handler function or to another message map. ATL defines another macro named `END_MSG_MAP` to mark the end of a message map. Between `BEGIN_MSG_MAP` and `END_MSG_MAP` lie some other macros for mapping window messages to member functions in the control. For example, here's a typical message map you might find in an ATL-based control:

```
BEGIN_MSG_MAP(CAFullControl)
    CHAIN_MSG_MAP(CComControl<CAFullControl>)
    DEFAULT_REFLECTION_HANDLER()
    MESSAGE_HANDLER(WM_TIMER, OnTimer);
    MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButton);
END_MSG_MAP()
```

This message map delegates most of the message processing to the control through the `CHAIN_MSG_MAP` macro and handles message reflection through the `DEFAULT_REFLECTION_HANDLER` macro. The message map also handles two window messages explicitly: `WM_TIMER` and `WM_LBUTTONDOWN`. These are standard window messages that are mapped using the `MESSAGE_HANDLER` macro. The macros simply produce a table relating window messages to member functions in the class. In addition to regular messages, message maps are capable of handling other sorts of events. Here's a rundown of the kinds of macros that can go in a message map.

Macro	Description
`MESSAGE_HANDLER`	Maps a Windows message to a handler function.
`MESSAGE_RANGE_HANDLER`	Maps a contiguous range of Windows messages to a handler function.
`COMMAND_HANDLER`	Maps a `WM_COMMAND` message to a handler function, based on the identifier and the notification code of the menu item, control, or accelerator.
`COMMAND_ID_HANDLER`	Maps a `WM_COMMAND` message to a handler function, based on the identifier of the menu item, control, or accelerator.
`COMMAND_CODE_HANDLER`	Maps a `WM_COMMAND` message to a handler function, based on the notification code.
`COMMAND_RANGE_HANDLER`	Maps a contiguous range of `WM_COMMAND` messages to a handler function, based on the identifier of the menu item, control, or accelerator.
`NOTIFY_HANDLER`	Maps a `WM_NOTIFY` message to a handler function, based on the notification code and the control identifier.
`NOTIFY_ID_HANDLER`	Maps a `WM_NOTIFY` message to a handler function, based on the control identifier.
`NOTIFY_CODE_HANDLER`	Maps a `WM_NOTIFY` message to a handler function, based on the notification code.
`NOTIFY_RANGE_HANDLER`	Maps a contiguous range of `WM_NOTIFY` messages to a handler function, based on the control identifier.

Table 3.

Handling messages within ATL works much the same as in MFC. ATL includes a single window procedure through which messages are routed. Technically, you can build your controls effectively without understanding everything about ATL's control architecture. However, this knowledge is sometimes helpful as you develop a control, and it's even more useful when debugging a control.

Developing the Control

Once the control is inserted into the server, you need to add some code to make the control do something. If you were to compile and load ATL's default control into a container, the results wouldn't be particularly interesting. You'd simply see a blank rectangle with the string "`ATL 3.0: Myatldiceob`." You'll want to add code to render the control, to represent the internal state of the control, to respond to events, and to generate events to send back to the container.

Deciding What to Draw

A good place to start working on a control is on its drawing code, you get instant gratification that way. This is a control that is visually represented by a couple of dice. The easiest way to render to the dice control is to draw bitmaps representing each of the six possible dice sides and then show the bitmaps on the screen. This implies that the dice control will maintain some variables to represent its state. For example, the control needs to manage the bitmaps for representing the dice as well as two numbers representing the first value shown by each die. Here is the code from **`MYATLDICEOBJ.H`** that represents the state of the dice:

```
#define MAX_DIEFACES 6
...
...
...
HBITMAP m_dieBitmaps[MAX_DIEFACES];
unsigned short m_nFirstDieValue;
unsigned short m_nSecondDieValue;
```

Before diving headfirst into the control's drawing code, you need to do a bit of preliminary work; the bitmaps need to be loaded. Presumably each die rendered by the dice control will show any one of six dice faces, so the control needs one bitmap for each face. Figure 56 shows what one of the dice bitmaps looks like.

Figure 56: A bitmap for the dice control.

If you draw the bitmaps one at a time, they'll have sequential identifiers in the **resource.h** file. Giving the bitmaps sequential identifiers will make them easier to load. Otherwise, you might need to modify the **resource.h** file, which contains the following identifiers:

```
#define IDB_DICE1      207
#define IDB_DICE2      208
#define IDB_DICE3      209
#define IDB_DICE4      210
#define IDB_DICE5      211
#define IDB_DICE6      212
```

Loading bitmaps is fairly straightforward. Cycle through the bitmap array, and load the bitmap resources. When they're stored in an array like this, grabbing the bitmap out of the array and showing it is much easier than if you didn't use an array. Here is the function that loads the bitmaps into the array:

```
BOOL Cmyatldiceob::LoadBitmaps()
{
    BOOL bSuccess = TRUE;

    for(int i=0; i<MAX_DIEFACES; i++)
    {
        DeleteObject(m_dieBitmaps[i]);
        m_dieBitmaps[i] = LoadBitmap(_Module.m_hInst,
MAKEINTRESOURCE(IDB_DICE1+i));
        if(!m_dieBitmaps[i])
        {
```

```
                ::MessageBox(NULL,
                "Failed to load bitmaps",
                NULL,
                MB_OK);
                bSuccess = FALSE;
            }
        }
        return bSuccess;
    }
```

The best place to call `LoadBitmaps()` is from within the control's constructor, as shown in the following code. To simulate a random roll of the dice, set the control's state so that the first and second die values are random numbers between 0 and 5 (these numbers will be used when the dice control is drawn):

```
    class Cmyatldiceob : // big inheritance list
    {
        Cmyatldiceob()
        {
            LoadBitmaps();
            srand((unsigned)time(NULL));
            m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
            m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
        }
```

Once the bitmaps are loaded, you'll want to render them. The dice control should include a function for showing each die face based on the current internal state of the dice. Here's where you first encounter ATL's drawing machinery.
One of the most convenient things about ATL-based controls (and MFC-based controls) is that all the drawing code happens in one place: within the control's `OnDraw()` function. `OnDraw()` is a virtual function of `COleControlBase`. Here's `OnDraw()`'s signature:

```
    virtual HRESULT OnDraw(ATL_DRAWINFO& di);
```

`OnDraw()` takes a single parameter: a pointer to an `ATL_DRAWINFO` structure. Among other things, the `ATL_DRAWINFO` structure contains a device context on which to render your control. Here's the `ATL_DRAWINFO` structure:

```
    struct ATL_DRAWINFO {
        UINT cbSize;
        DWORD dwDrawAspect;
        LONG lindex;
        DVTARGETDEVICE* ptd;
        HDC hicTargetDev;
        HDC hdcDraw;
        LPCRECTL prcBounds; //Rectangle in which to draw
        LPCRECTL prcWBounds; //WindowOrg and Ext if metafile
        BOOL bOptimize;
        BOOL bZoomed;
        BOOL bRectInHimetric;
        SIZEL ZoomNum;      //ZoomX = ZoomNum.cx/ZoomNum.cy
        SIZEL ZoomDen;
    };
```

As you can see, there's a lot more information here than a simple device context. While you can count on the framework filling it out correctly for you, it's good to know where the information in the structure comes from and how it fits into the picture.
ActiveX Controls are interesting because they are drawn in **two contexts**. The **first** and most obvious context is when the control is active and it draws within the actual drawing space of the client. The other, less-obvious context in which controls are drawn is during **design time** (as when an ActiveX control resides in a Visual Basic form in design mode). In the first context, ActiveX Controls render themselves to a live screen device context. In the second context, ActiveX Controls render themselves to a metafile device context.

Many (though not all) ATL-based controls are composed of at least one window. So ActiveX Controls need to render themselves during the `WM_PAINT` message. Once the control receives the `WM_PAINT` message, the message routing architecture passes control to `CComControlBase::OnPaint`. Remember, `CComControlBase` is one of the control's base classes. `CComControlBase::OnPaint` performs several steps. The function begins by creating a painting device context (using `BeginPaint()`). Then `OnPaint()` creates an `ATL_DRAWINFO` structure on the stack and initializes the fields within the structure. `OnPaint()` sets up `ATL_DRAWINFO` to show the entire content (the `dwDrawAspect` field is set to `DVASPECT_CONTENT`). `OnPaint()` also sets the `lindex` field to `_1`, sets the drawing device context to the newly created painting device context, and sets up the bounding rectangle to be the client area of the control's window. Then `OnPaint()` goes on to call `OnDrawAdvanced()`. The default `OnDrawAdvanced()` function prepares a normalized device context for drawing. You can override this method if you want to use the device context passed by the container without normalizing it. ATL then calls your control class's `OnDraw()` method.

The second context in which the `OnDraw()` function is called is when the control draws on to a metafile. The control draws itself on to a metafile whenever someone calls `IViewObjectEx::Draw`. `IViewObjectEx` is one of the interfaces implemented by the ActiveX control. ATL implements the `IViewObjectEx` interface through the template class `IViewObjectExImpl`. `IViewObjectExImpl::Draw` is called whenever the control needs to take a snapshot of its presentation space for the container to store. In this case, the container creates a metafile device context and hands it to the control. `IViewObjectExImpl` puts an `ATL_DRAWINFO` structure on the stack and initializes. The bounding rectangle, the index, the drawing aspect, and the device contexts are all passed in as parameters by the client. The rest of the drawing is the same in this case, the control calls `OnDrawAdvanced()`, which in turn calls your version of `OnDraw()`.

Once you're armed with this knowledge, writing functions to render the bitmaps becomes fairly straightforward. To show the first die face, create a memory-based device context, select the object into the device context, and BitBlt the memory device context into the real device context. Here's the code:

```
void Cmyatldiceob::ShowFirstDieFace(ATL_DRAWINFO& di)
{
    BITMAP bmInfo;
    GetObject(m_dieBitmaps[m_nFirstDieValue-1], sizeof(bmInfo),
&bmInfo);

    SIZE size;

    size.cx = bmInfo.bmWidth;
    size.cy = bmInfo.bmHeight;

    HDC hMemDC;
    hMemDC = CreateCompatibleDC(di.hdcDraw);

    HBITMAP hOldBitmap;
    HBITMAP hbm = m_dieBitmaps[m_nFirstDieValue-1];
    hOldBitmap = (HBITMAP)SelectObject(hMemDC, hbm);

    if (hOldBitmap == NULL)
        return; // destructors will clean up

    BitBlt(di.hdcDraw,
           di.prcBounds->left+1,
           di.prcBounds->top+1,
           size.cx,
           size.cy,
           hMemDC, 0,
           0,
           SRCCOPY);

    SelectObject(di.hdcDraw, hOldBitmap);
    DeleteDC(hMemDC);
}
```

Showing the second die face is more or less the same process; just make sure that the dice are represented separately. For example, you probably want to change the call to `BitBlt()` so that the two dice bitmaps are shown side by side.

```
void Cmyatldiceob::ShowSecondDieFace(ATL_DRAWINFO& di)
{
    // This code is exactly the same as ShowFirstDieFace
    //  except the second die is positioned next to the first die.
    BitBlt(di.hdcDraw,
           di.prcBounds->left+size.cx + 2,
           di.prcBounds->top+1,
           size.cx,
           size.cy,
           hMemDC, 0,
           0, SRCCOPY);
    // The rest is the same as in ShowFirstDieFace
}
```

The last step is to call these two functions whenever the control is asked to render itself, during the control's `OnDraw()` function. `ShowFirstDieFace()` and `ShowSecondDieFace()` will show the correct bitmap based on the state of `m_nFirstDieValue` and `m_nSecondDieValue`:

```
HRESULT OnDraw(ATL_DRAWINFO& di)
{
      RECT& rc = *(RECT*)di.prcBounds;

      ShowFirstDieFace(di);
      ShowSecondDieFace(di);
      return S_OK;
}
```

At this point, if you compile and load this control into some ActiveX Control container (like a Visual Basic form or an MFC-based dialog), you'll see two die faces staring back at you. Now it's time to add some code to enliven the control and roll the dice.

Responding to Window Messages

Just looking at two dice faces isn't that much fun. You want to make the dice work. A good way to get the dice to appear to jiggle is to use a timer to generate events and then respond to the timer by showing a new pair of dice faces. Setting up a Windows timer in the control means adding a function to handle the timer message and adding a macro to the control's message map. Start by using ClassView to add a handler for `WM_TIMER`. Right-click on the `Cmyatldiceob` symbol in ClassView, and select **Add Windows Message Handler** from the context menu. This adds a prototype for the `OnTimer()` function and an entry into the message map to handle the `WM_TIMER` message. Add some code to the `OnTimer()` function to handle the `WM_TIMER` message. The `OnTimer()` function should look like the code shown below.

```
LRESULT Cmyatldiceob::OnTimer(UINT msg, WPARAM wParam, LPARAM lParam,
BOOL& bHandled)
{
    if(m_nTimesRolled > 15)
    {

        m_nTimesRolled = 0;
        KillTimer(1);
    } else {
        m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
        m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
        FireViewChange();
        m_nTimesRolled++;
    }
```

```
    bHandled = TRUE;
    return 0;
}
```

This function responds to the timer message by generating two random numbers, setting up the control's state to reflect these two new numbers, and then asking the control to refresh itself by calling `FireViewChange()`. Notice the function kills the timer as soon as the dice have rolled a certain number of times. Also notice that the message handler tells the framework that it successfully handled the function by setting the `bHandled` variable to `TRUE`.
Notice there's an entry for `WM_TIMER` in the control's message map. Because `WM_TIMER` is just a plain vanilla window message, it's represented with a standard `MESSAGE_HANDLER` macro as follows:

```
BEGIN_MSG_MAP(Cmyatldiceob)
    CHAIN_MSG_MAP(CComControl<Cmyatldiceob>)
    DEFAULT_REFLECTION_HANDLER()
    MESSAGE_HANDLER(WM_TIMER, OnTimer);
END_MSG_MAP()
```

As you can tell from this message map, the dice control already handles the gamut of Windows messages through the `CHAIN_MSG_MAP` macro. However, now the pair of dice has the ability to simulate rolling by responding to the timer message. Setting a timer causes the control to repaint itself with a new pair of dice numbers every quarter of a second or so. Of course, there needs to be some way to start the dice rolling. Because this is an ActiveX control, it's reasonable to allow client code to start rolling the dice via a call to a function in one of its incoming interfaces. Use ClassView to add a `RollDice()` function to the main interface. Do this by right-clicking on the `IMyatldiceobj` interface appearing in ClassView on the left side of the screen and selecting **Add Method** from the pop up menu. Then add a `RollDice()` function. Microsoft Visual C++ adds a function named `RollDice()` to your control. Implement `RollDice()` by setting the timer for a reasonably short interval and then returning `S_OK`. Add the following code:

```
STDMETHODIMP Cmyatldiceob::RollDice()
{
    SetTimer(1, 250);
    return S_OK;
}
```

If you load the dice into an ActiveX control container, you'll now be able to browse and call the control's methods and roll the dice.
In addition to using the incoming interface to roll the dice, the user might reasonably expect to roll the dice by double-clicking the control. To enable this behavior, just add a message handler to trap the mouse-button-down message by adding a function to handle a left-mouse double click.

```
LRESULT Cmyatldiceob::OnLButtonDblClick(UINT uMsg,
                                        WPARAM wParam,
                                        LPARAM lParam,
                                        BOOL& bHandled)
{
    RollDice();
    bHandled = TRUE;
    return 0;
}
```

Then be sure you add an entry to the message map to handle the `WM_LBUTTONDOWN` message:

```
BEGIN_MSG_MAP(Cmyatldiceob)
    // ...
    // Other message handlers
    // ...
    MESSAGE_HANDLER(WM_LBUTTONDBLCLK, OnLButtonDblClick)
END_MSG_MAP()
```

When you load the dice control into a container and double-click on it, you should see the dice roll. Now that you've added rendering code and given the control the ability to roll, it's time to add some properties.

Adding Properties and Property Pages

You've just seen that ActiveX controls have an external presentation state. The presentation state is the state reflected when the control draws itself. In addition, most ActiveX controls also have an internal state. The control's internal state is a set of variables exposed to the outside world via interface functions. These internal variables are also known as properties.
For example, imagine a simple grid implemented as an ActiveX control. The grid has an external presentation state and a set of internal variables for describing the state of the grid. The properties of a grid control would probably include the number of rows in the grid, the number of columns in the grid, the color of the lines composing the grid, the type of font used, and so forth.
As you saw in Module 28, adding properties to an ATL-based class means adding member variables to the class and then using ClassWizard to create get and put functions to access these properties. For example, two member variables that you might add to the dice control include the dice color and the number of times the dice are supposed to roll before stopping. Those two properties could easily be represented as a pair of short integers as shown here:

```
class ATL_NO_VTABLE Cmyatldiceob :
...
...
{
    ...
    ...
    short m_nDiceColor;
    short m_nTimesToRoll;
    ...
    ...
};
```

To make these properties accessible to the client, you need to add `get` and `put` functions to the control. Right-clicking on the interface symbol in ClassView brings up a context menu, giving you a choice to **Add Property**, which will present you with the option of adding these functions. Adding `DiceColor()` and `TimesToRoll()` properties to the control using ClassView will add four new functions to the control: `get_DiceColor()`, `put_DiceColor()`, `get_TimesToRoll()`, and `put_TimesToRoll()`. The `get_DiceColor()` function should retrieve the state of `m_nDiceColor`:

```
STDMETHODIMP Cmyatldiceob::get_DiceColor(short * pVal)
{
    *pVal = m_nDiceColor;
    return S_OK;
}
```

To make the control interesting, `put_DiceColor()` should change the colors of the dice bitmaps and redraw the control immediately. This example uses red and blue dice as well as the original black and white dice. To make the control show the new color bitmaps immediately after the client sets the dice color, the `put_DiceColor()` function should load the new bitmaps according to new color, and redraw the control:

```
STDMETHODIMP Cmyatldiceob::put_DiceColor(short newVal)
{
    if(newVal < 3 && newVal >= 0)
        m_nDiceColor = newVal;
    LoadBitmaps();
    FireViewChange();
    return S_OK;
}
```

Of course, this means that `LoadBitmaps()` needs to load the bitmaps based on the state of `m_nDiceColor`, so we need to add the following code to our existing `LoadBitmaps()` function:

```
BOOL Cmyatldiceob::LoadBitmaps()
{
    int i;
    BOOL bSuccess = TRUE;
    int nID = IDB_DICE1;

    switch(m_nDiceColor)
    {
        case 0:
            nID = IDB_DICE1;
            break;

        case 1:
            nID = IDB_BLUEDICE1;
            break;

        case 2:
            nID = IDB_REDDICE1;
            break;
    }

    for(i=0; i<MAX_DIEFACES; i++)
    {
        DeleteObject(m_dieBitmaps[i]);
        m_dieBitmaps[i] = LoadBitmap(_Module.m_hInst,
MAKEINTRESOURCE(nID+i));
        if(!m_dieBitmaps[i])
        {
            ::MessageBox(NULL,
            "Failed to load bitmaps",
            NULL, MB_OK);
            bSuccess = FALSE;
        }
    }
    return bSuccess;
}
```

Just as the dice color property reflects the color of the dice, the number of times the dice rolls should be reflected by the state of the `TimesToRoll` property. The `get_TimesToRoll()` function needs to read the `m_nTimesToRoll` member, and the `put_TimesToRoll()` function needs to modify `m_nTimesToRoll`. Add code shown below.

```
STDMETHODIMP Cmyatldiceob::get_TimesToRoll(short * pVal)
{
    *pVal = m_nTimesToRoll;
    return S_OK;
}

STDMETHODIMP Cmyatldiceob::put_TimesToRoll(short newVal)
{
    m_nTimesToRoll = newVal;
    return S_OK;
}
```

Finally, instead of hard-coding the number of times the dice rolls, use the `m_nTimesToRoll` variable to determine when to kill the timer.

```
LRESULT Cmyatldiceob::OnTimer(UINT msg, WPARAM wParam, LPARAM lParam,
BOOL& bHandled)
{
    if(m_nTimesRolled > m_nTimesToRoll)
```

```
    {
        m_nTimesRolled = 0;
        KillTimer(1);
        Fire_DiceRolled(m_nFirstDieValue, m_nSecondDieValue);
        if(m_nFirstDieValue == m_nSecondDieValue)
            Fire_Doubles(m_nFirstDieValue);
        if(m_nFirstDieValue == 1 && m_nSecondDieValue == 1)
            Fire_SnakeEyes();
    } else {
        m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
        m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
FireViewChange();
        m_nTimesRolled++;
    }

    bHandled = TRUE;
    return 0;
}
```

Now these two properties are exposed to the outside world. When the client code changes the color of the dice, the control loads a new set of bitmaps and redraws the control with the new dice faces. When the client code changes the number of times to roll, the dice control uses that information to determine the number of times the dice control should respond to the `WM_TIMER` message. So the next question is, "How are these properties accessed by the client code?" One way is through a control's property pages.

Property Pages

Since ActiveX controls are usually UI gadgets meant to be mixed into much larger applications, they often find their homes within places such as Visual Basic forms and MFC form views and dialogs. When a control is instantiated, the client code can usually reach into the control and manipulate its properties by calling certain functions on the control's incoming interface functions. However, when an ActiveX control is in design mode, accessing the properties through the interface functions isn't always practical. It would be unkind to tool developers to force them to go through the interface functions all the time just to tweak some properties in the control. Why should the tool vendor who is creating the client have to provide UI for managing control properties? That's what property pages are for. Property pages are sets of dialogs implemented by the control for manipulating properties. That way, the tool vendors don't have to keep re-creating dialog boxes for tweaking the properties of an ActiveX control.

How Property Pages Are Used. Property pages are usually used in one of two ways. The first way is through the control's `IOleObject` interface. The client can call `IOleObject`'s `DoVerb()` function, passing in the properties verb identifier (named `OLEIVERB_PROPERTIES` and defined as the number -7) to ask the control to show its property pages. The control then displays a dialog, or property frame, that contains all the control's property pages. For example, Figure 57 shows the **Property Pages** dialog containing the property pages for the Microsoft FlexGrid 6.0 control.

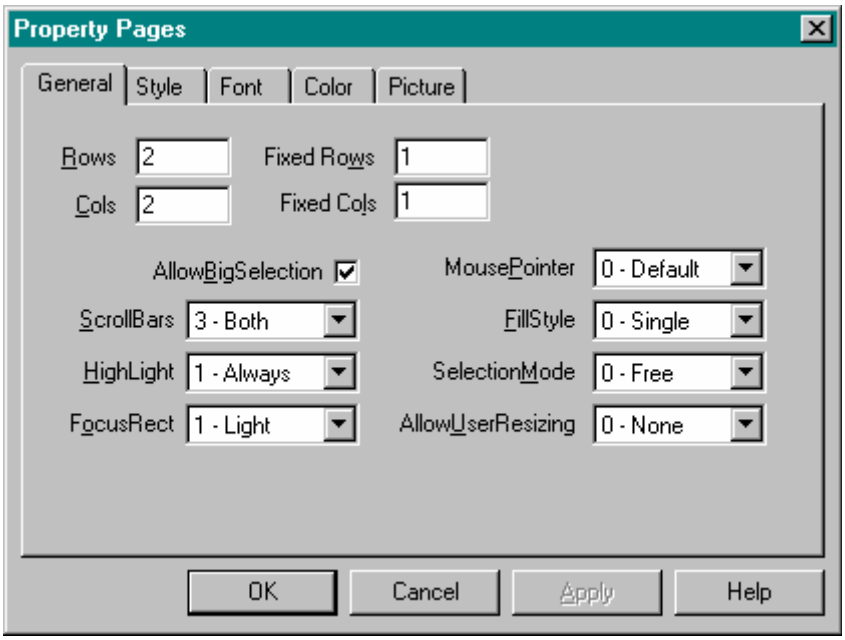

Figure 57: The Microsoft FlexGrid 6.0 control executing the properties verb.

Property pages are a testament to the power of COM. As it turns out, each single property page is a separate COM object (named using a `GUID` and registered like all the other COM classes on your system). When a client asks an ActiveX control to show its property pages via the properties verb, the control passes its own list of property page `GUID`s into a system API function named `OleCreatePropertyFrame()`. `OleCreatePropertyFrame()` enumerates the property page `GUID`s, calling `CoCreateInstance()` for each property page. The property frame gets a copy of an interface so that the frame can change the properties within the control. `OleCreatePropertyFrame()` calls back to the control when the user clicks the **OK** or **Apply** button.

The second way clients use property pages is when the client asks the control for a list of property page `GUID`s. Then the client calls `CoCreateInstance()` on each property page and installs each property page in its own frame. Figure 58 shows an example of how Visual C++ uses the Microsoft FlexGrid property pages in its own property dialog frame.

This second method is by far the most common way for a control's property pages to be used. Notice that the property sheet in Figure 58 contains a **General** tab in addition to the control's property pages, and that the General tab shown in Figure 55 has been renamed to the **Control** tab. The General property page in Figure 58 belongs to Visual C++. The **Control**, **Style**, **Font**, **Color**, and **Picture** property pages belong to the control (even though they're being shown within the context of Visual C++).

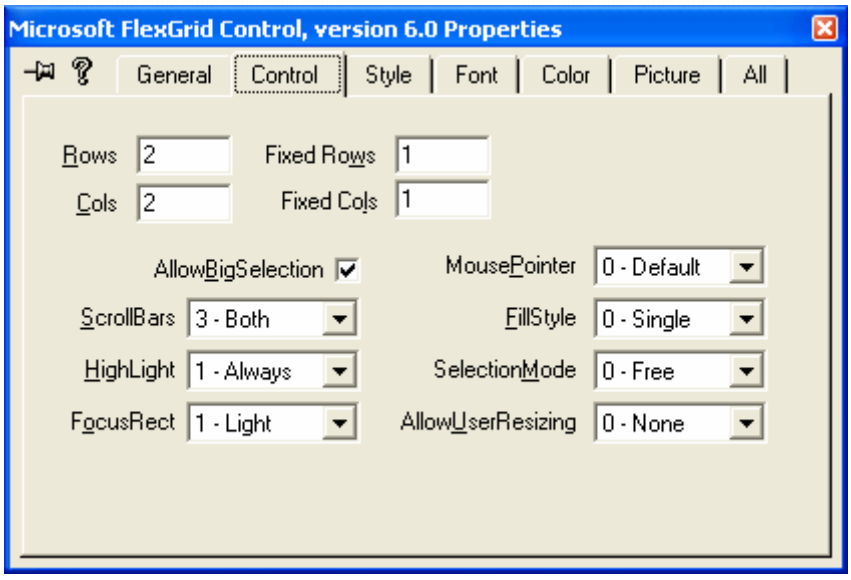

Figure 58: Microsoft Visual C++ inserting the Microsoft FlexGrid 6.0 property pages into its own dialog box for editing resource properties.

For a property page to work correctly, the control that the property page is associated with needs to implement `ISpecifyPropertyPages` and the property page object needs to implement an interface named `IPropertyPage`. With this in mind, let's examine exactly how ATL implements property pages.

Adding a Property Page to Your Control. You can use the Visual Studio **ATL Object Wizard** to create property pages in your ATL project. To create a property page, perform the following steps:

1. Select **New ATL Object** from the Visual C++ **Insert** menu.
2. From the **ATL Object Wizard** dialog, select **Controls** from the **Category** list.
3. Select **Property Page** from the **Objects** list.
4. Click **Next**.
5. Fill in the required information on the **ATL Object Wizard Properties** dialog, and click **OK**.

ATL's Object Wizard generates a dialog template and includes it as part of a control's resources. In the dice control example, the two properties you're concerned with are the **color of the dice** and the **number of times to roll the dice**. The dialog template created by ATL's Object Wizard is **blank**, so you'll want to add a couple of controls to represent these properties. In this example, the user will be able to select the dice color from a **combo box** and enter the number of times the dice should roll in an **edit control**, as shown in Figure 59.

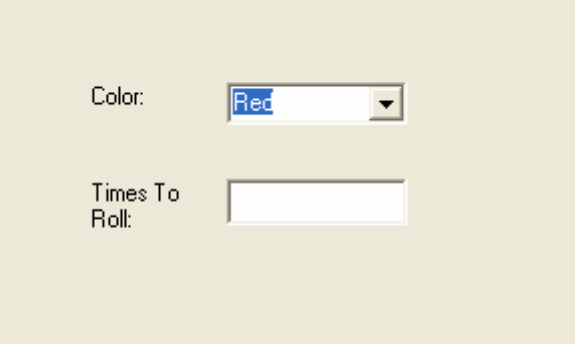

Figure 59. The property page dialog template.

The ATL Object Wizard also creates a C++ class for you that implement the interface necessary for the class to behave as a property page. In addition to generating this C++ class, the ATL Object Wizard makes the class part of the project. The ATL Object Wizard adds the new property page class to the IDL file within the coclass section. In addition, the ATL Object Wizard appends the property page to the object map so that `DllGetClassObject()` can find the property page class. Finally, the ATL Object Wizard adds a new **Registry script** so that the DLL makes the correct Registry entries when the control is registered. Here is the header file created by the ATL Object Wizard for a property page named `DiceMainPropPage`:

```
#include "resource.h"  // main symbols

class ATL_NO_VTABLE CDiceMainPropPage :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CDiceMainPropPage, &CLSID_DiceMainPropPage>,
    public IPropertyPageImpl<CDiceMainPropPage>,
    public CDialogImpl<CDiceMainPropPage>
{
public:
    CDiceMainPropPage()
    {
        m_dwTitleID = IDS_TITLEDiceMainPropPage;
        m_dwHelpFileID = IDS_HELPFILEDiceMainPropPage;
        m_dwDocStringID = IDS_DOCSTRINGDiceMainPropPage;
    }
```

```
    enum {IDD = IDD_DICEMAINPROPPAGE};

DECLARE_REGISTRY_RESOURCEID(IDR_DICEMAINPROPPAGE)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CDiceMainPropPage)
    COM_INTERFACE_ENTRY(IPropertyPage)
END_COM_MAP()

BEGIN_MSG_MAP(CDiceMainPropPage)
    CHAIN_MSG_MAP(IPropertyPageImpl<CDiceMainPropPage>)
END_MSG_MAP()

STDMETHOD(Apply)(void)
{
    ATLTRACE(_T("CDiceMainPropPage::Apply\n"));
    for (UINT i = 0; i < m_nObjects; i++)
    {
        // Do something interesting here
        // ICircCtl* pCirc;
        // m_ppUnk[i]->QueryInterface(IID_ICircCtl, (void**)&pCirc);
        // pCirc->put_Caption(CComBSTR("something special"));
        // pCirc->Release();
    }
    m_bDirty = FALSE;
    return S_OK;
}
};
```

Examining this property page listing reveals that ATL's property page classes are composed of several ATL templates: `CComObjectRootEx` (to implement `IUnknown`), `CComCoClass` (the class object for the property page), `IPropertyPageImpl` (for implementing `IPropertyPage`), and `CDialogImpl` for implementing the dialog-specific behavior.

As with most other COM classes created by ATL's Object Wizard, most of the code involved in getting a property page to work is boilerplate code. Notice that besides the constructor and some various maps, the only other function is one named **Apply**.

Before getting into the mechanics of implementing a property page, it's helpful to take a moment to understand how the property page architecture works. The code you need to type in to get the property pages working will then make more sense.

When the client decides it's time to show some property pages, a modal dialog frame needs to be constructed. The frame is constructed by either the client or by the control itself. If the property pages are being shown via the `DoVerb()` function, the control constructs the frame. If the property pages are being shown within the context of another application, as when Visual C++ shows the control's property pages within the IDE, the client constructs the dialog frame. The key to the dialog frame is that it holds property page sites (small objects that implement `IPropertyPageSite`) for each property page.

The client code (the modal dialog frame, in this case) then enumerates through a list of `GUID`s, calling `CoCreateInstance()` on each one of them and asking for the `IPropertyPage` interface. If the COM object produced by `CoCreateInstance()` is a property page, it implements the `IPropertyPage` interface. The dialog frame uses the `IPropertyPage` interface to talk to the property page. Here's the declaration of the `IPropertyPage` interface:

```
interface IPropertyPage : public IUnknown
{
    HRESULT SetPageSite(IPropertyPageSite *pPageSite) = 0;
    HRESULT Activate(HWND hWndParent,
                     LPCRECT pRect,
                     BOOL bModal) = 0;
    HRESULT Deactivate( void) = 0;
```

```
        HRESULT GetPageInfo(PROPPAGEINFO *pPageInfo) = 0;
        HRESULT SetObjects(ULONG cObjects,
                           IUnknown **ppUnk) = 0;
        HRESULT Show(UINT nCmdShow) = 0;
        HRESULT Move(LPCRECT pRect) = 0;
        HRESULT IsPageDirty(void) = 0;
        HRESULT Apply(void) = 0;
        HRESULT Help(LPCOLESTR pszHelpDir) = 0;
        HRESULT TranslateAccelerator(MSG *pMsg) = 0;
    };
```

Once a property page has been created, the property page and the client code need some channels to communicate back and forth with the control. After the property dialog frame successfully calls `QueryInterface()` for `IPropertyPage` on the property page objects, the frame calls `IPropertyPage::SetPageSite` on each `IPropertyPage` interface pointer it holds, passing in an `IPropertyPageSite` interface pointer. The property page sites within the property frame provide a way for each property page to call back to the frame. The property page site provides information to the property page and receives notifications from the page when changes occur. Here's the `IPropertyPageSite` interface:

```
    interface IPropertyPageSite : public IUnknown
    {
        public:
            virtual HRESULT OnStatusChange(DWORD dwFlags) = 0;
            virtual HRESULT GetLocaleID(LCID *pLocaleID) = 0;
            virtual HRESULT GetPageContainer(IUnknown *ppUnk) = 0;
            virtual HRESULT TranslateAccelerator(MSG *pMsg) = 0;
    };
```

In addition to the frame and control connecting to each other through `IPropertyPage` and `IPropertyPageSite`, each property page needs a way to talk back to the control. This is usually done when the dialog frame calls `IPropertyPage::SetObjects`, passing in the control's `IUnknown`. Figure 60 illustrates the property page architecture.

Now that you see how ActiveX Control property pages work in general, understanding how they work within ATL will be a lot easier. You'll see how ATL's property pages work, in cases when the client code exercises the control's properties verb as well as in cases when environments like Visual C++ integrate a control's property pages into the IDE.

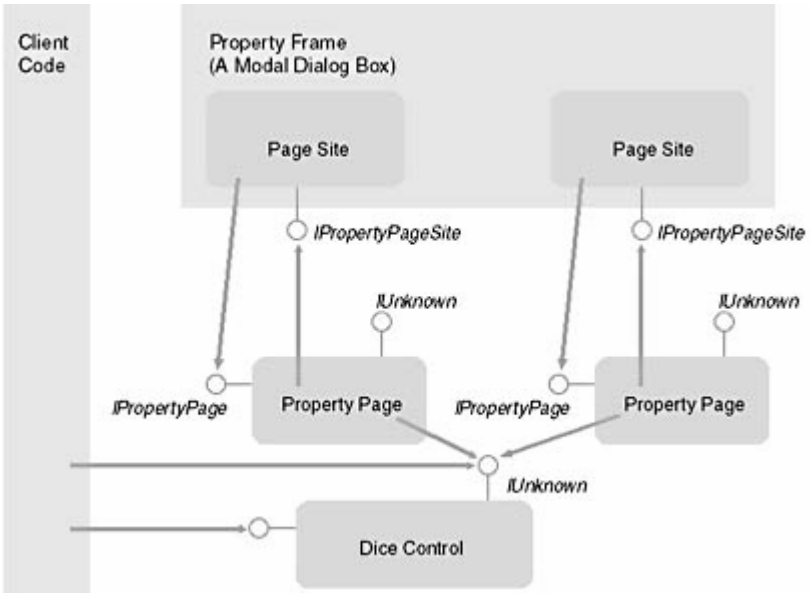

Figure 60: How the property pages, the property frame, and the property page sites communicate.

ATL and the Properties Verb. The first way in which an ActiveX control shows its property pages is when the client invokes the properties verb by calling `IOleObject::DoVerb` using the constant `OLEIVERB_PROPERTIES`. When the client calls `DoVerb()` in an ATL-based control, the call ends up in the function `CComControlBase::DoVerbProperties`, which simply calls `OleCreatePropertyFrame()`, passing in its own `IUnknown` pointer and the list of property page `GUID`s. `OleCreatePropertyFrame()` takes the list of `GUID`s, calling `CoCreateInstance()` on each one to create the property pages, and arranges them within the dialog frame. `OleCreatePropertyFrame()` uses each property page's `IPropertyPage` interface to manage the property page, as described in "**How Property Pages Are Used**"

ATL Property Maps. Of course, understanding how `OleCreatePropertyFrame()` works from within the ATL-based control begs the next question: where does the list of property pages actually come from? ATL uses macros to generate lists of property pages called property maps. Whenever you add a new property page to an ATL-based control, you need to set up the list of property pages through these macros. ATL includes several macros for implementing property maps: `BEGIN_PROPERTY_MAP`, `PROP_ENTRY`, `PROP_ENTRY_EX`, `PROP_PAGE`, and `END_PROPERTY_MAP`. Here are those macros in the raw:

```
struct ATL_PROPMAP_ENTRY
{
    LPCOLESTR szDesc;
    DISPID dispid;
    const CLSID* pclsidPropPage;
    const IID* piidDispatch;
    DWORD dwOffsetData;
    DWORD dwSizeData;
    VARTYPE vt;
};

#define BEGIN_PROPERTY_MAP(theClass) \
    typedef _ATL_PROP_NOTIFY_EVENT_CLASS __ATL_PROP_NOTIFY_EVENT_CLASS;
\
        typedef theClass _PropMapClass; \
    static ATL_PROPMAP_ENTRY* GetPropertyMap()\
    {\
        static ATL_PROPMAP_ENTRY pPropMap[] = \
    {

#define PROP_PAGE(clsid) \
    {NULL, NULL, &clsid, &IID_NULL},

#define PROP_ENTRY(szDesc, dispid, clsid) \
    {OLESTR(szDesc), dispid, &clsid, &IID_IDispatch},

#define PROP_ENTRY_EX(szDesc, dispid, clsid, iidDispatch) \
    {OLESTR(szDesc), dispid, &clsid, &iidDispatch},

#define END_PROPERTY_MAP() \
        {NULL, 0, NULL, &IID_NULL} \
    }; \
    return pPropMap; \
}
```

When you decide to add property pages to a COM class using ATL's property page macros, according to the ATL documentation you should put these macros into your class's header file. For example, if you want to add property pages to the dice control, you'd add the following code to the C++ class:

```
class ATL_NO_VTABLE Cmyatldiceob :
 ...
 ...
 ...
{
```

```
    ...
    ...
    BEGIN_PROP_MAP(Cmyatldiceob)
        PROP_ENTRY("Caption goes here…", 2,
                   CLSID_MainPropPage)
        PROP_ENTRY_EX("Caption goes here…", 3,
                      CLSID_SecondPropPage,
                      DIID_SecondDualInterface)
        PROP_PAGE(CLSID_StockColorPage)
    END_PROPERTY_MAP()
};
```

ATL's property map macros set up the list of GUIDs representing property pages. ATL's property maps are composed of an array of ATL_PROPMAP_ENTRY structures. The BEGIN_PROPERTY_MAP macro declares a static variable of this structure. The PROP_PAGE macro inserts a GUID into the list of property pages. PROP_ENTRY inserts a property page GUID into the list as well as associating a specific control property with the property page. The final macro, PROP_ENTRY_EX, lets you associate a certain dual interface to a property page. When client code invokes the control's properties verb, the control just rips through this list of GUIDs and hands the list over to the OleCreatePropertyFrame() so that the property can create the property pages.

Property Pages and **Development Tools Executing** the properties verb isn't the only way for an ActiveX control to show its property pages. As we mentioned before, folks who write tools such as Visual Basic and Visual C++ might want programmatic access to a control's property pages. For example, when using MFC to work on a dialog box containing an ActiveX control, right-clicking on the control to view the properties gives you a dialog frame produced by Visual C++ (as opposed to the dialog frame produced by OleCreatePropertyFrame()). Visual C++ uses the control's ISpecifyPropertyPages interface to get the list of GUIDs (the list generated by the property page macros). Here's the ISpecifyPropertyPages interface definition:

```
interface ISpecifyPropertyPages : public IUnknown
{
    HRESULT GetPages(CAUUID *pPages);
};

typedef struct tagCAUUID
{
    ULONG     cElems;
    GUID FAR* pElems;
} CAUUID;
```

ATL implements the ISpecifyPropertyPages::GetPages function by cycling through the list of GUIDS (produced by the property map macros) and returning them within the CAUUID structure. Environments like Visual C++ use each GUID in a call to CoCreateInstance() to create a new property page. The property page site and the property page exchange interfaces. The property page site holds on to the property page's IPropertyPage interface, and the property page holds on to the property site's IPropertyPageSite interface. After the dialog frame constructs the property pages, it needs to reflect the current state of the ActiveX control through the dialog controls. For that you need to override the property page's Show() method.

Showing the Property Page. The property page's Show() method is called whenever the property page is about to be shown. A good thing for a property page to do at this time is fetch the values from the ActiveX control and populate the property page's controls. Remember that the property page holds on to an array of unknown pointers (they're held in the IPropertyPageImpl's m_ppUnk array.) To access the ActiveX control's properties, you need to call QueryInterface() on the unknown pointers and ask for the interface that exposes the properties. In this case, the interface is IMyatldiceobj. Once the property page has the interface, it can use the interface to fetch the properties and plug the values into the dialog box controls. Here's the overridden Show() method:

```
#include "atldicesrvr.h"

class ATL_NO_VTABLE CDiceMainPropPage :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CDiceMainPropPage, &CLSID_DiceMainPropPage>,
```

```
    public IPropertyPageImpl<CDiceMainPropPage>,
    public CDialogImpl<CDiceMainPropPage>
{
    ...
    ...
    ...
STDMETHOD(Show)( UINT nCmdShow )
{
        HRESULT hr;

        USES_CONVERSION;

        if(nCmdShow == SW_SHOW ||
           nCmdShow == SW_SHOWNORMAL) {
            for (UINT i = 0; i < m_nObjects; i++)
            {
                CComQIPtr< IATLDieceObj, &IID_IATLDieceObj >
pMyatldiceob(m_ppUnk[i]);
                short nColor = 0;

                if FAILED(pMyatldiceob->get_DiceColor(&nColor))
                {
                    CComPtr<IErrorInfo> pError;
                    CComBSTR            strError;
                    GetErrorInfo(0, &pError);
                    pError->GetDescription(&strError);
                    MessageBox(OLE2T(strError), _T("Error"),
MB_ICONEXCLAMATION);
                    return E_FAIL;
                }
                HWND hWndComboBox = GetDlgItem(IDC_COLOR);
                ::SendMessage(hWndComboBox,
                              CB_SETCURSEL,
                              nColor, 0);

                short nTimesToRoll = 0;
                if FAILED(
                    pMyatldiceob->get_TimesToRoll(&nTimesToRoll))
                {
                    CComPtr<IErrorInfo> pError;
                    CComBSTR            strError;
                    GetErrorInfo(0, &pError);
                    pError->GetDescription(&strError);
                    MessageBox(OLE2T(strError),
                        _T("Error"), MB_ICONEXCLAMATION);
                    return E_FAIL;
                }
                SetDlgItemInt(IDC_TIMESTOROLL, nTimesToRoll, FALSE);
            }
        }
        m_bDirty = FALSE;
        hr = IPropertyPageImpl<CDiceMainPropPage>::Show(nCmdShow);
        return hr;
    }
};
```

In addition to adding code to prepare to show the dialog box, you need to add code allowing users to set the control's properties. Whenever the user changes a property, the property dialog activates the **Apply** button, indicating that the user can apply the newly set properties. When the user presses the Apply button, control jumps to the property page's Apply function so you need to insert some code in here to make the Apply button work.

Handling the Apply Button. After the user finishes manipulating the properties, he or she clicks either the Apply button or the OK button to save the changes. In response, the client code asks the property page to apply the new properties to the control. Remember that the ActiveX control and the property page are separate COM objects, so they need to communicate via interfaces. Here's how the process works.
When you create a property page using the ATL Object Wizard, ATL overrides the `Apply()` function from `IPropertyPage` for you. The property page site uses this function for notifying the property page of changes that need to be made to the control. When the property page's `Apply()` function is called, it's time to synch up the state of the property page with the state of the control. Remember, the control's `IUnknown` interface was passed into the property page early in the game via a call to `IPropertyPage::SetObjects`. The interface pointers are stored in the property page's `m_ppUnk` array. Most property pages respond to the `Apply()` function by setting the state of the ActiveX control properties through the interface provided. In the case of our example ATL-based property page, this means examining the value in the combo box and the edit box and setting the new values inside the control itself, like this:

```
#include "atldicesrvr.h"

class ATL_NO_VTABLE CDiceMainPropPage :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CDiceMainPropPage, &CLSID_DiceMainPropPage>,
    public IPropertyPageImpl<CDiceMainPropPage>,
    public CDialogImpl<CDiceMainPropPage>
{
    ...
    ...
    STDMETHOD(Apply)(void)
    {
        USES_CONVERSION;
        ATLTRACE(_T("CDiceMainPropPage::Apply\n"));
        for (UINT i = 0; i < m_nObjects; i++)
        {
            CComQIPtr<IATLDieceObj,
                      &IID_IATLDieceObj> pMyatldiceob(m_ppUnk[i]);
            HWND hWndComboBox = GetDlgItem(IDC_COLOR);
            short nColor = (short)::SendMessage(hWndComboBox,
                                                CB_GETCURSEL,
                                                0, 0);
            if(nColor >= 0 && nColor <= 2)
            {
                if FAILED(pMyatldiceob->put_DiceColor(nColor))
                {
                    CComPtr<IErrorInfo> pError;
                    CComBSTR            strError;
                    GetErrorInfo(0, &pError);
                    pError->GetDescription(&strError);
                    MessageBox(OLE2T(strError),
                          _T("Error"),
                          MB_ICONEXCLAMATION);
                    return E_FAIL;
                }
            }

            short nTimesToRoll = (short)GetDlgItemInt(IDC_TIMESTOROLL);
            if FAILED(pMyatldiceob->put_TimesToRoll(nTimesToRoll))
            {
                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;
    }
```

Property Persistence

Once you have added properties to the control, it's logical that you might want to have those properties persist with their container. For example, imagine Hasbro buys your dice control to include in its new Windows version of Monopoly. The game vendor uses your dice control within one of the Monopoly dialog boxes and configures the control so that the dice are blue and they roll 23 times before stopping. If the dice control had a sound property, the Monopoly authors could configure the dice to emit a beep every time they roll. When someone plays the game and rolls the dice, that person will see a pair of blue dice that roll 23 times before stopping and they will hear the dice make a sound while they roll. Remember that these properties are all properties of the control. If you're using the control in an application, chances are good you'll want these properties to be saved with the application.
Fortunately, adding persistence support to your control is almost free when you use the ATL property macros. You've already seen how to add the property pages to the control DLL using the property map macros. As it turns out, these macros also make the properties persistent.
You can find ATL's code for handling the persistence of a control's properties within the `CComControlBase` class. `CComControlBase` has a member function named `IPersistStreamInit_Save()` that handles saving a control's properties to a stream provided by the client. Whenever the container calls `IPersistStreamInit::Save`, ATL ends up calling `IPersistStreamInit_Save()` to do the actual work. `IPersistStreamInit_Save()` works by retrieving the control's property map, the list of properties maintained by the control. Remember that the `BEGIN_PROPERTY_MAP` macro adds a function named `GetPropertyMap()` to the control. The first item written out by `IPersistStreamInit_Save()` is the control's extents (its size on the screen). `IPersistStreamInit_Save()` then cycles through the property map to write the contents of the property map out to the stream. For each property, the control calls `QueryInterface()` on itself to get its own dispatch interface. As `IPersistStreamInit_Save()` goes through the list of properties, the control calls `IDispatch::Invoke` on itself to get the property based on the `DISPID` associated with the property. The property's `DISPID` is included as part of the property map structure. The property comes back from `IDispatch::Invoke` as a `Variant`, and `IPersistStreamInit_Save()` writes the property to the stream provided by the client.

Bidirectional Communication (Events)

Now that the dice control has properties and property pages and renders itself to a device context, the last thing to do is to add some events to the control. Events provide a way for the control to call back to the client code and inform the client code of certain events as they occur.
For example, the user can roll the dice. Then when the dice stop rolling, the client application can fish the dice values out of the control. However, another way to implement the control is to set it up so that the control notifies the client application when the dice have rolled using an event. Here you'll see how to add some events to the dice control. We'll start by understanding how ActiveX Control events work.

How Events Work. When a control is embedded in a container (such as a Visual Basic form or an MFC-based dialog box), one of the steps the client code takes is to establish a connection to the control's event set. That is, the client implements an interface that has been described by the control and makes that interface available to the control. That way, the control can talk back to the container.
Part of developing a control involves defining an interface that the control can use to call back to the client. For example, if you're developing the control using MFC, ClassWizard will define the interface and produce some functions you can call from within the control to fire events back to the client. If you're developing the control in ATL, you can accomplish the same result by defining the event callback interface in the control's IDL and using ClassView to create a set of callback proxy functions for firing the events to the container. When the callback interface is defined by the control, the container needs to implement that interface and hand it over to the control. The client and the control do this through the `IConnectionPointContainer` and `IConnectionPoint` interfaces.
`IConnectionPointContainer` is the interface that a COM object implements to indicate that it supports connections. `IConnectionPointContainer` represents a collection of connections available to the client.

Within the context of ActiveX Controls, one of these connections is usually the control's main event set. Here's the `IConnectionPointContainer` interface:

```
interface IConnectionPointContainer : IUnknown
{
    HRESULT FindConnectionPoint(REFIID riid, IConnectionPoint **ppcp) =
0;
    HRESULT EnumConnectionPoints(IEnumConnectionsPoint **ppec) = 0;
};
```

`IConnectionPointContainer` represents a collection of `IConnectionPoint` interfaces. Here's the `IConnectionPoint` interface:

```
interface IConnectionPoint : IUnknown
{
    HRESULT GetConnectionInterface(IID *pid) = 0;
    HRESULT GetConnectionPointContainer(IConnectionPointContainer
**ppcpc) = 0;
    HRESULT Advise(IUnknown *pUnk, DWORD *pdwCookie) = 0;
    HRESULT Unadvise(dwCookie) = 0;
    HRESULT EnumConnections(IEnumConnections **ppec) = 0;
}
```

The container creates the control by calling `CoCreateInstance()` on the control. As the control and the container are establishing the interface connections between themselves, one of the interfaces the container asks for is `IConnectionPointContainer` (that is, the container calls `QueryInterface()` asking for `IID_IConnectionPointContainer`). If the control supports connection points (the control answers "Yes" when queried for `IConnectionPointContainer`), the control uses `IConnectionPointContainer::FindConnectionPoint` to get the `IConnectionPoint` interface representing the main event set. The container knows the `GUID` representing the main event set by looking at the control's type information as the control is inserted into the container.
If the container can establish a connection point to the control's main event set (that is, `IConnectionPointContainer::FindConnectionPoint` returns an `IConnectionPoint` interface pointer), the container uses `IConnectionPoint::Advise` to subscribe to the callbacks. Of course, to do this the container needs to implement the callback interface defined by the control (which the container can learn about by using the control's type library). Once the connection is established, the control can call back to the container whenever the control fires off an event. Here's what it takes to make events work within an ATL-based ActiveX control.

Adding Events to the Dice Control. There are several steps to adding event sets to your control. Some of them are hidden by clever wizardry. First, use IDL to describe the events. Second, add a proxy that encapsulates the connection points and event functions. Finally, fill out the control's connection map so that the client and the object have a way to connect to each other. Let's examine each step in detail.
When using ATL to write an ActiveX control, IDL is the place to start adding events to your control. The event callback interface is described within the IDL so the client knows how to implement the callback interface correctly. The IDL is compiled into a type library that the client will use to figure out how to implement the callback interface. For example, if you wanted to add events indicating the dice were rolled, doubles were rolled, and snake eyes were rolled, you'd describe the callback interface like this in the control's IDL file:

```
library ATLDICESRVRLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(21C85C43-0BFF-11d1-8CAA-FD10872CC837),
        helpstring("Events created from rolling dice")
    ]
    dispinterface _IMyatldiceobjEvents
    {
```

```
    properties:
    methods:
        [id(1)] void DiceRolled([in]short x, [in] short y);
        [id(2)] void Doubles([in] short x);
        [id(3)] void SnakeEyes();
}

[
    uuid(6AED4EBD-0991-11D1-8CAA-FD10872CC837),
    helpstring("Myatldiceob Class")
]
coclass Myatldiceob
{
    [default] interface IATLDieceObj;
    [default, source] dispinterface _IMyatldiceobjEvents;
};
```

The control's callback interface is defined as a dispatch interface (note the dispinterface keyword) because that's the most generic kind of interface available. When it comes to callback interfaces, most environments understand only `IDispatch`. The code on the previous page describes a callback interface to be implemented by the client (if the client decides it wants to receive these callbacks). We added this dice events interface by hand. The Object Wizard will put one in for you. It might have a different name than the one we have listed. For example, the Wizard is likely to put in an interface named `IATLObjEvents`.

Implementing the Connection Point. After you've described the callback interface within the IDL and compiled the control, the control's type information will contain the callback interface description so that the client will know how to implement the callback interface. However, you don't yet have a convenient way to fire these events from the control. You could, of course, call back to the client by setting up calls to `IDispatch::Invoke` by hand. However, a better way to do this is to set up a proxy (a set of functions wrapping calls to `IDispatch`) to handle the hard work for you. To generate a set of functions that you can call to fire events in the container, use the **Implement Connection Point** menu option from ClassView.
In ClassView, click the right mouse button while the cursor is hovering over the `Cmyatldiceob` symbol. This brings up the context menu for the `Cmyatldiceob` item. Choose **Implement Connection Point** from the menu to bring up the **Implement Connection Point** dialog box. This dialog box asks you to locate the type information describing the interface you expect to use when calling back to the container (the `_IMyatldiceobjEvents` interface, in this case). By default, this dialog box looks at your control's type library. The dialog box reads the type library and shows the interfaces found within it. Choose `_IMyatldiceobjEvents` and click **OK**. Doing so creates a C++ class that wraps the dice events interface. Given the above interface definition, here's the code generated by the Implement Connection Point dialog box:

```
template <class T>
class CProxy_IATLDieceObjEvents :
public IConnectionPointImpl<T, &DIID__IATLDieceObjEvents,
CComDynamicUnkArray>
{
    //Warning this class may be recreated by the wizard.
public:
};

{
    //Warning this class may be recreated by the wizard.
public:
    VOID Fire_Doubles(SHORT x)
    {
        T* pT = static_cast<T*>(this);
        int nConnectionIndex;
        CComVariant* pvars = new CComVariant[1];
        int nConnections = m_vec.GetSize();

        for (nConnectionIndex = 0;
```

```
         nConnectionIndex < nConnections;
         nConnectionIndex++)
    {
        pT->Lock();
        CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
        pT->Unlock();
        IDispatch* pDispatch =
            reinterpret_cast<IDispatch*>(sp.p);
        if (pDispatch != NULL)
        {
            pvars[0].vt = VT_I2;
            pvars[0].iVal= x;
            DISPPARAMS disp = { pvars, NULL, 1, 0 };
            pDispatch->Invoke(0x1, IID_NULL,
                              LOCALE_USER_DEFAULT,
                              DISPATCH_METHOD, &disp,
                              NULL, NULL, NULL);
        }
    }
    delete[] pvars;
}
VOID Fire_DiceRolled(SHORT x, SHORT y)
{
    T* pT = static_cast<T*>(this);
    int nConnectionIndex;
    CComVariant* pvars = new CComVariant[2];
    int nConnections = m_vec.GetSize();

    for (nConnectionIndex = 0;
         nConnectionIndex < nConnections;
         nConnectionIndex++)
    {
        pT->Lock();
        CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
        pT->Unlock();
        IDispatch* pDispatch =
            reinterpret_cast<IDispatch*>(sp.p);
        if (pDispatch != NULL)
        {
            pvars[1].vt = VT_I2;
            pvars[1].iVal= x;
            pvars[0].vt = VT_I2;
            pvars[0].iVal= y;
            DISPPARAMS disp = { pvars, NULL, 2, 0 };
            pDispatch->Invoke(0x2, IID_NULL,
                              LOCALE_USER_DEFAULT,
                              DISPATCH_METHOD, &disp,
                              NULL, NULL, NULL);
        }
    }
    delete[] pvars;

}

VOID Fire_SnakeEyes()
{
    T* pT = static_cast<T*>(this);
    int nConnectionIndex;
    int nConnections = m_vec.GetSize();

    for (nConnectionIndex = 0;
```

```
                nConnectionIndex < nConnections;
                nConnectionIndex++)
            {
                pT->Lock();
                CComPtr<IUnknown> sp = m_vec.GetAt(nConnectionIndex);
                pT->Unlock();
                IDispatch* pDispatch =
                    reinterpret_cast<IDispatch*>(sp.p);
                if (pDispatch != NULL)
                {
                    DISPPARAMS disp = { NULL, NULL, 0, 0 };
                    pDispatch->Invoke(0x3, IID_NULL,
                                      LOCALE_USER_DEFAULT,
                                      DISPATCH_METHOD, &disp,
                                      NULL, NULL, NULL);
                }
            }

    }
};
```

The C++ class generated by the connection point generator serves a dual purpose. First, it acts as the specific connection point. Notice that it derives from `IConnectionPointImpl`. Second, the class serves as a proxy to the interface implemented by the container. For example, if you want to call over to the client and tell the client that doubles were rolled, you'd simply call the proxy's `Fire_Doubles()` function. Notice how the proxy wraps the `IDispatch` call so that you don't have to get your hands messy dealing with variants by yourself.

Establishing the Connection and Firing the Events. The final step in setting up the event set is to add the connection point to the dice control and turn on the `IConnectionPointContainer` interface. The connection point dialog box added the `CProxy_IMyatldiceobjEvents` class to the dice control's inheritance list, which provides the `IConnectionPoint` implementation inside the control. An ATL class named `IConnectionPointContainerImpl` provides the implementation of `IConnectionPointContainer`. These two interfaces should be in the dice control's inheritance list like this:

```
class Cmyatldiceob :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CStockPropImpl<Cmyatldiceob, IATLDieceObj,
                          &IID_IATLDieceObj,
                          &LIBID_ATLDICESRVRLib>,
    public CComControl<Cmyatldiceob>,
    public IPersistStreamInitImpl<Cmyatldiceob>,
    public IOleControlImpl<Cmyatldiceob>,
    public IOleObjectImpl<Cmyatldiceob>,
    public IOleInPlaceActiveObjectImpl<Cmyatldiceob>,
    public IViewObjectExImpl<Cmyatldiceob>,
    public IOleInPlaceObjectWindowlessImpl<Cmyatldiceob>,
    public IConnectionPointContainerImpl<Cmyatldiceob>,
    public IPersistStorageImpl<Cmyatldiceob>,
    public ISpecifyPropertyPagesImpl<Cmyatldiceob>,
    public IQuickActivateImpl<Cmyatldiceob>,
    public IDataObjectImpl<Cmyatldiceob>,
    public IProvideClassInfo2Impl<&CLSID_Myatldiceob,
                                  &DIID__IMyatldiceobjEvents,
                                  &LIBID_ATLDICESRVRLib>,
    public IPropertyNotifySinkCP<Cmyatldiceob>,
    public CComCoClass<Cmyatldiceob, &CLSID_Myatldiceob>,
    public CProxy_DDiceEvents< Cmyatldiceob >
{
  ...
  ...
  ...
```

```
};
```

Having these classes in the inheritance list inserts the machinery in your control that makes connection points work. Whenever you want to fire an event to the container, all you need to do is call one of the functions in the proxy. For example, a good time to fire these events is from within the control's `OnTimer()` method, firing a `DiceRolled()` event whenever the timer stops, firing a `SnakeEyes()` event whenever both die faces have the value 1, and firing a `Doubles()` event when both die faces are equal:

```
Cmyatldiceob::OnTimer(UINT msg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled)
{
    if(m_nTimesRolled > m_nTimesToRoll)
    {
        m_nTimesRolled = 0;
        KillTimer(1);
        Fire_DiceRolled(m_nFirstDieValue, m_nSecondDieValue);
        if(m_nFirstDieValue == m_nSecondDieValue)
            Fire_Doubles(m_nFirstDieValue);
        if(m_nFirstDieValue == 1 &&
           m_nSecondDieValue == 1)
            Fire_SnakeEyes();
    } else {
        m_nFirstDieValue = (rand() % (MAX_DIEFACES)) + 1;
        m_nSecondDieValue = (rand() % (MAX_DIEFACES)) + 1;
        FireViewChange();
        m_nTimesRolled++;
    }
        bHandled = TRUE;
        return 0;
}
```

Finally, notice the connection map contains entries for the control's connection points:

```
BEGIN_CONNECTION_POINT_MAP(Cmyatldiceob)
    CONNECTION_POINT_ENTRY(DIID__IMyatldiceobjEvents)
    CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
END_CONNECTION_POINT_MAP()
```

The control uses this map to hand back connection points as the client requests them.

Using the Control

So, how do you use the control once you've written it? The beauty of COM is that as long as the client and the object agree on their shared interfaces, they don't need to know anything else about each other. All the interfaces implemented within the dice control are well understood by a number of programming environments. You've already seen how to use ActiveX Controls within an MFC-based dialog box. The control you just wrote will work fine within an MFC-based dialog box, just use the **Add To Project** menu option under the **Project** menu. Select **Registered ActiveX Controls** and insert the Myatldiceob component into your project. Visual C++ will read the dice control's type information and insert all the necessary COM glue to make the dialog box and the control talk together. This includes all the OLE embedding interfaces as well as the connection and event interfaces. In addition, you could just as easily use this control from within a Visual Basic form. When working on a Visual Basic project, select **References** from the **Project** menu and insert the dice control into the Visual Basic project.

Conclusion

ActiveX Controls are one of the **most widely used applications of COM in the real world today**. To summarize, ActiveX controls are **just COM objects** that happen to implement a number of standard interfaces that environments like Visual C++ and Visual Basic understand how to use. These interfaces deal with rendering, persistence, and events, allowing you to drop these components into the aforementioned programming environments and use them right away.

In the past, MFC was the only practical way to implement ActiveX Controls. However, these days ATL provides a reasonable way of implementing ActiveX Controls, provided you're willing to follow ATL's rules. For example, if you buy into the ATL architecture for writing controls, you'll have to dip down into Windows and start working with window handles and device context handles in their raw forms. However, the tradeoff is often worthwhile, because ATL provides more flexibility when developing ActiveX controls. For example, dual interfaces are free when using ATL, whereas they're a real pain to implement in MFC.

----------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.