white paper

DEVELOPMENTOR℠

WTL Makes UI Programming a Joy, Part 1: The Basics

The Windows Template Library (WTL) is available on the January 2000 Platform SDK. It is an SDK sample produced by members of the ATL (Active Template Library) team chartered with building an ATL-based wrapper around the windowing portions of the Win32 API. Since version 2.0, ATL has had simple windowing wrapper classes, such as CWindow, CWindowImpl, and CDialogImpl. However, when compared with MFC, ATL's windowing classes were little more than a tease. Even in ATL 3.0, there is no support for such popular features as MDI, command bars, DDX, printing, GDI, or even a port of the most beloved class in all of MFC, CString. Without these features WTL cannot satisfy the overwhelming majority of MFC programmers. WTL is what members of the ATL team think a windowing framework should be. **Table 1** shows the list of features that WTL provides as compared to MFC.

Table 1: MFC vs. WTL

Feature	MFC	WTL
Stand-alone library	Yes	No (built on ATL)
AppWizard support	Yes	Yes
ClassWizard support	Yes	No
Officially supported by Microsoft	Yes	No (Supported by volunteers inside MS)
Support for OLE Documents	Yes	No
Support for Views	Yes	Yes
Support for Documents	Yes	No
Basic Win32 & Common Control Wrappers	Yes	Yes
Advanced Common Control Wrappers (Flat scrollbar, IP Address, Pager Control, etc.)	No	Yes
Command Bar support (including bitmapped context menus)	No (MFC does provide dialog bars)	Yes
CString	Yes	Yes
GDI wrappers	Yes	Yes
Helper classes (CRect, Cpoint, etc.)	Yes	Yes
Property Sheets/Wizards	Yes	Yes
SDI, MDI support	Yes	Yes
Multi-SDI support	No	Yes
MRU Support	Yes	Yes
Docking Windows/Bars	Yes	No
Splitters	Yes	Yes
DDX	Yes	Yes (not as extensive as MFC)
Printing/Print Preview	Yes	Yes
Scrollable Views	Yes	Yes
Custom Draw/Owner Draw Wrapper	No	Yes

Dharma Shukla, Chris Sells,
and Nenad Stefanovic

white paper
DEVELOPMENTOR™
www.develop.com

Feature	MFC	WTL
Message/Command Routing	Yes	Yes
Common Dialogs	Yes	Yes
HTML Views	Yes	Yes
Single Instance Applications	No	No
UI Updating	Yes	Yes
Template-based	No	Yes
Size of a *statically* linked do-nothing SDI application with toolbar, status bar, and menu	228KB + MSVCRT.DLL (288KB)	24k (with /OPT:NOWIN98) (+ MSVCRT.DLL if you use CString)
Size of a *dynamically* linked do-nothing SDI application with toolbar, status bar, and menu	24KB + MFC42.DLL (972KB) + MSVCRT.DLL (288KB)	N/A
Runtime Dependencies	CRT (+ MFC42.DLL, if dynamically linked)	None (CRT if you use CString)

WTL certainly doesn't do everything that MFC does. MFC supports classic OLE, the doc/view architecture and docking windows; WTL does not. In addition, the lack of "official" support from Microsoft is of concern. However, the "unofficial" support from past members of the ATL team (those same members who got WTL included as an SDK sample) and the active ATL community should mitigate the support issue. Why is the ATL community likely to embrace the WTL? The last four items in the feature list tell the tale: WTL is template-based, the minimum application size is small (24KB) and there are no DLL dependencies (except the CRT if you use CString). In short, the flexibility and small footprint that we know and love about ATL has been carried forward into WTL. Coupled with this, WTL is a programming model very similar to that of MFC and it includes an MFC port of CString.

In this two part series, we unravel the mysteries of WTL. In Part 1, we wet our feet in the details of the WTL frame windows architecture. We explain how to write WTL based SDI, MDI, multi-SDI, and Explorer style apps. Next, we cover the WTL helper classes including DDX wrappers. Finally, we look at the WTL AppWizard and the sample applications that ship with WTL.

Part 2 of our series covers the details of WTL command bar architecture, common controls wrappers, and custom UI widgets. We delve into WTL's message routing architecture, including message cracking, filtering, and idle handling. Our journey to explore WTL won't be complete unless we cover common dialog boxes, property pages and sheets, printing support, and scrolling windows – all of which we plan for part 2.

Before we show you how to build applications WTL, let’s review how to build applications in raw ATL.

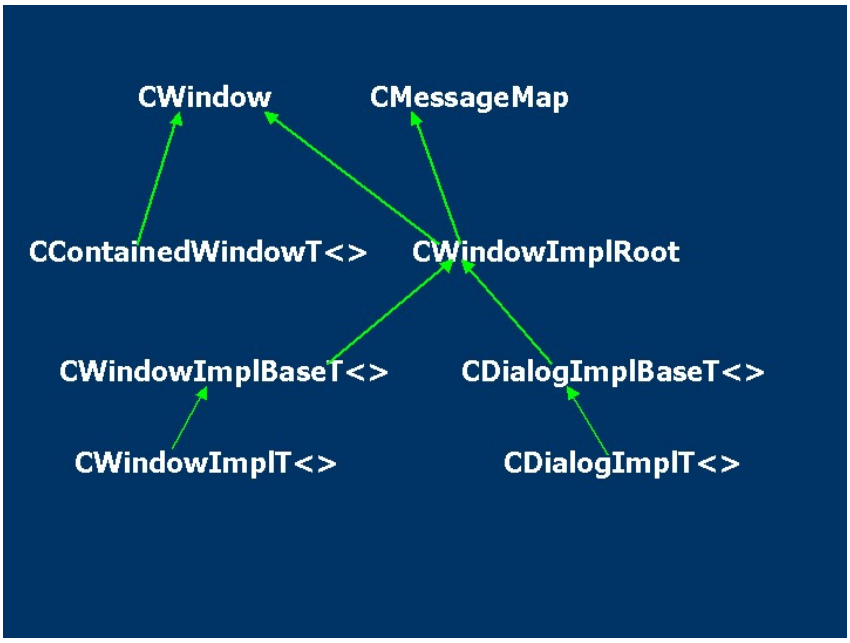
Dharma Shukla, Chris Sells,
and Nenad Stefanovic

white paper
DEVELOPMENTOR™
www.develop.com

ATL, the Foundation of WTL

ATL provides a set of classes for windowing. Originally aimed to support ATL's COM control and OLE Property Page architectures, these classes also form the basis for WTL. ATL provides all the basic windowing functional aspects, including window/dialog creation and management, windows procedures, message routing, windows subclassing, superclassing, and message chaining. **Figure 1** shows the ATL windowing-class hierarchy.

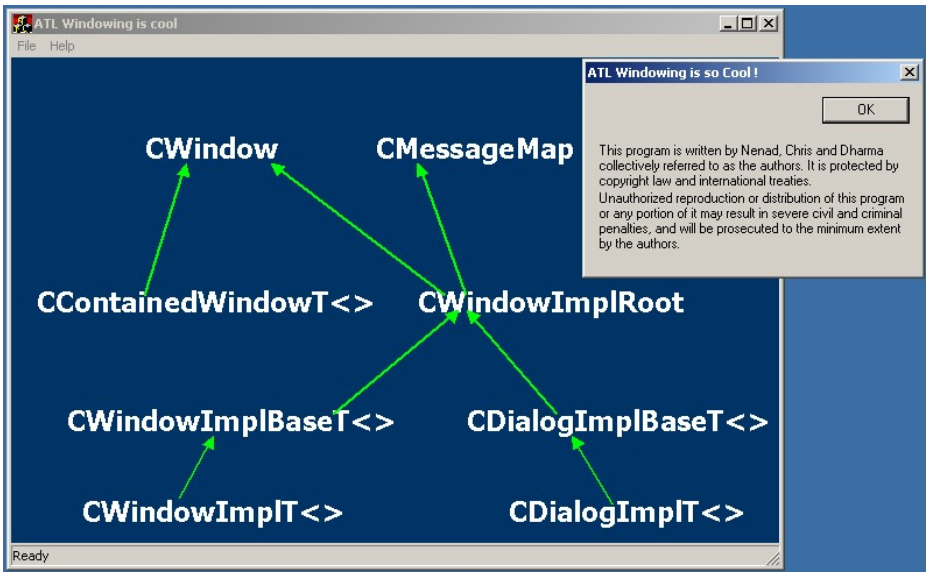
Figure 1: ATL Windowing-class Hierarchy



(名字后面带个 T 的,表示这个类是模板类)

To create a window or a dialog in straight (直接)ATL, you need to derive(继承) from `CWindowImpl` or `CDialogImpl`, respectively(分别). To give you an idea of how to create a window and a dialog with ATL, **Figure 2** provides a simple SDI application developed using the windowing support provided by ATL.

Figure 2: Simple SDI Application



The application consists of an SDI frame window that contains a menu, status bar, and a client area. It also provides an About box to demonstrate how to use dialogs in raw ATL. We started this application by creating a simple Win32 Application project in VC6 and adding the minimal support to do ATL windowing. **Figure 3** shows the main source files.

Figure 3: Main Source Files

```
//--------------------------------------------------------------------
// stdafx.h :

#if !defined(AFX_STDAFX_H__DE06DA2F_25B6_41BA_9B20_A17362C38C8C__INCLUDED_)
#define AFX_STDAFX_H__DE06DA2F_25B6_41BA_9B20_A17362C38C8C__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define STRICT
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif
#define _ATL_APARTMENT_THREADED

#include <atlbase.h>
//You may derive a class from CComModule and use it if you want to override
//something, but do not change the name of _Module
```

```
extern CComModule _Module;
#include <atlwin.h>

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_STDAFX_H__DE06DA2F_25B6_41BA_9B20_A17362C38C8C__INCLUDED)


//--------------------------------------------------------------------
//MainFrame.H

#pragma once
#ifndef _MAINFRAME_H_
#define _MAINFRAME_H_

#include "commctrl.H"

class CMainFrame : public CWindowImpl<CMainFrame,CWindow,
CWinTraits<WS_OVERLAPPEDWINDOW|WS_CLIPCHILDREN , WS_EX_APPWINDOW | WS_EX_WINDOWEDGE> >
{
public:
        CMainFrame():m_hWndStatusBar(NULL),m_hBmp(NULL) {
                m_hBmp =
                        LoadBitmap(_Module.GetResourceInstance(),
                                        MAKEINTRESOURCE(IDB_ATLWINDOWING));
        }
        ~CMainFrame() {
                if(m_hBmp) {
                        ::DeleteObject(m_hBmp);
                        m_hBmp = NULL;
                }
        }
        BEGIN_MSG_MAP(CMainFrame)
                MESSAGE_HANDLER(WM_PAINT, OnPaint)
                MESSAGE_HANDLER(WM_CREATE, OnCreate)
                MESSAGE_HANDLER(WM_SIZE, OnSize)
                MESSAGE_HANDLER(WM_CLOSE, OnClose)
                COMMAND_ID_HANDLER(ID_FILE_EXIT, OnFileExit)
                COMMAND_ID_HANDLER(ID_APP_ABOUT, OnAbout)
        END_MSG_MAP()

        void OnFinalMessage(HWND /*hWnd*/) {}
        LRESULT OnCreate(UINT, WPARAM wParam, LPARAM lParam, BOOL& bHandled) {
                DefWindowProc();
                m_hWndStatusBar = ::CreateStatusWindow(WS_CHILD | WS_VISIBLE |
                                        WS_CLIPCHILDREN | WS_CLIPSIBLINGS | SBARS_SIZEGRIP,
                                        _T("Ready"), m_hWnd, 1);

                return 0L;
        }
        LRESULT OnSize(UINT, WPARAM wParam, LPARAM lParam, BOOL& bHandled) {
                DefWindowProc();
                ::SendMessage(m_hWndStatusBar, WM_SIZE, 0, 0);
                return 0L;
        }
        LRESULT OnPaint(UINT, WPARAM, LPARAM, BOOL&) {
```

```
                PAINTSTRUCT ps;
                HDC         hdc = BeginPaint(&ps);

                RECT    rect; GetClientRect(&rect);

                //Bitmap
                HDC hDCMem =  ::CreateCompatibleDC(hdc);
                HBITMAP hBmpOld = (HBITMAP)::SelectObject(hDCMem,m_hBmp);

                BITMAP bmp;
                ::GetObject(m_hBmp, sizeof(BITMAP), &bmp);
                SIZE size = { bmp.bmWidth,bmp.bmHeight };

                ::BitBlt(hdc, rect.left, rect.top, (size.cx), (size.cy), hDCMem,
                        0,0,SRCCOPY);

                //cleanup
                ::SelectObject(hDCMem,hBmpOld);
                ::DeleteDC(hDCMem);
                hDCMem = NULL;

                EndPaint(&ps);
                return 0;
        }
        LRESULT OnClose(UINT,WPARAM,LPARAM, BOOL&) {
                DestroyWindow();
                PostQuitMessage(0);
                return 0L;
        }
        LRESULT OnFileExit(WORD,WORD wID, HWND,BOOL&) {
                SendMessage(WM_CLOSE);
                return 0L;
        }
        LRESULT OnAbout(WORD, WORD wID, HWND, BOOL&) {
                CAboutDialog dlg;
                dlg.DoModal();
                return 0L;
        }

private:
        struct CAboutDialog : public CDialogImpl<CAboutDialog> {
                enum { IDD = IDD_ABOUT };
                BEGIN_MSG_MAP(CAboutDialog)
                        COMMAND_ID_HANDLER(IDOK, OnClose)
                END_MSG_MAP()

                LRESULT OnClose(WORD, WORD wID, HWND, BOOL&) {
                        EndDialog(wID);
                        return 0L;
                }
        };

        HWND            m_hWndStatusBar;
        HBITMAP         m_hBmp;
};
#endif

//---------------------------------------------------------------------
```

//AtlHelloWindowing.Cpp

```
#include "stdafx.h"
#include "resource.h"
#include "MainFrame.H"

CComModule _Module;

extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance,HINSTANCE, LPTSTR lpCmdLine, int nShowCmd)
{
      lpCmdLine = GetCommandLine(); //this line necessary for
_ATL_MIN_CRT

      _Module.Init(0, hInstance,NULL);

      //Load Common Controls
      INITCOMMONCONTROLSEX iccx;
      iccx.dwSize = sizeof(iccx);
      iccx.dwICC =  ICC_WIN95_CLASSES  |ICC_BAR_CLASSES ;
      ::InitCommonControlsEx(&iccx);


      HMENU hMenu = LoadMenu(_Module.GetResourceInstance(),
                        MAKEINTRESOURCE(IDR_MAINFRAME));

      CMainFrame wndFrame;
      wndFrame.GetWndClassInfo().m_wc.hIcon= ::LoadIcon(
                                    _Module.GetResourceInstance(),
                                          MAKEINTRESOURCE(IDI_FORM));
      wndFrame.GetWndClassInfo().m_wc.style = 0;//CS_HREDRAW|CS_VREDRAW;

      wndFrame.Create(GetDesktopWindow(), CWindow::rcDefault,
                  _T("ATL Windowing is cool"), 0, 0, (UINT)hMenu);

      wndFrame.ShowWindow(nShowCmd);
      wndFrame.UpdateWindow();

      MSG msg;
      while (GetMessage(&msg, 0, 0, 0)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      _Module.Term();
      return 0;
}
```

In `MainFrame.h`, `CMainFrame` stores two data members – the `HWND` of the statusbar control and the `HBITMAP` of a bitmap. `CMainFrame` creates the statusbar as a child window in the `WM_CREATE` handler by calling the `CreateStatusWindow` function from `comctl32.dll`, and takes care of sizing the statusbar in its `WM_SIZE` processing. The `WM_PAINT` handler uses the bitmap to demonstrate a little bit of GDI (Graphic Device Interface). ATL doesn't provide any wrappers for drawing, so the painting is all raw Win32 calls.

While the `MESSAGE_HANDLER` macro requires you to do your own message cracking (notice the `WPARAM` and `LPARAM` parameters of most of the message handlers), ATL does crack both `WM_COMMAND` and `WM_NOTIFY` messages for you via a family of macros. `MainFrame.h` shows the use of `COMMAND_ID_HANDLER`, which `CMainFrame` uses to handle its menu commands.

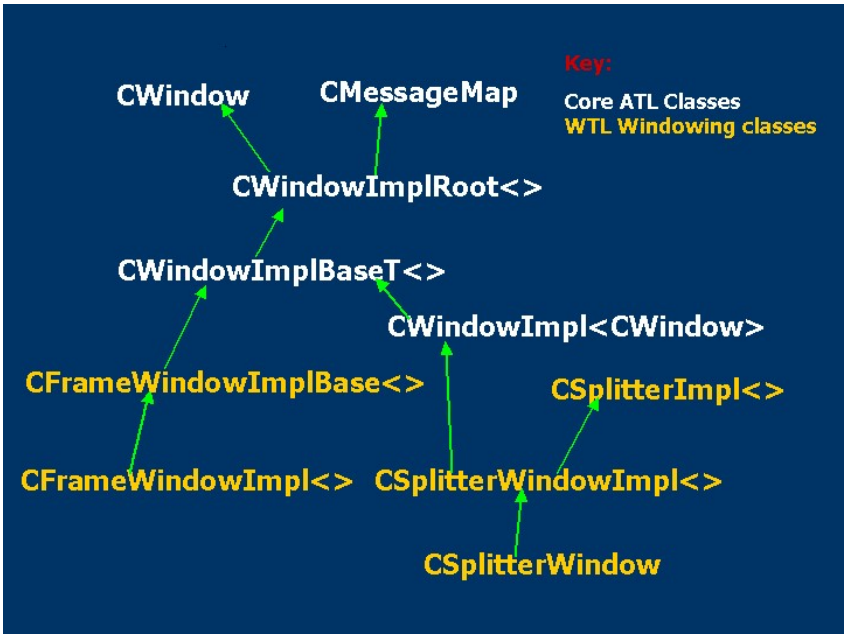
Finally, `CAboutDialog` is a dialog that derives from `CDialogImpl`. The `OnAbout` menu handler invokes it with a call to `DoModal`.

HelloAtlWindowing.cpp shows `_tWinMain`, the entry point of our ATL application. It creates an instance of `CMainFrame` and calls the `Create`, `ShowWindow`, and `UpdateWindow` member functions, which wraps the Win32 `CreateWindow`, `ShowWindow`, and `UpdateWindow` functions, respectively. Finally, because this is a Windows application after all, we pump the messages via a standard `GetMessage-DispatchMessage` loop. Those of you familiar with Win32 will notice the lack of calls to `RegisterClass` and the window procedures, but will recognize pretty much everything else. Those of you familiar with MFC might be surprised that ATL doesn't provide such basic niceties as automatic support for creating status bars and menus (not to mention toolbars) and for pumping messages. While it's obvious that ATL helps, it doesn't go nearly far enough. Let's not forget that ATL's roots are firmly grounded in COM. The windowing support it provides is a by-product and not the main purpose of its existence. ATL allows but does not support MDI, Multi-SDI, or Explorer-style applications, nor does it provide common control wrappers, DDX, or GDI helpers. For those, we turn to WTL.

The WTL Way

The WTL class that makes it possible for us to more easily create an SDI frame window is `CFrameWindowImpl`. (**Figure 4** shows WTL's SDI windowing hierarchy.)

Figure 4: WTL's SDI Windowing Hierarchy



The `CFrameWindowImpl` class derives from `CFrameWindowImplBase` and thereby inherits all the functionality of a standard application frame window. The following lists features a WTL created frame window provides support for:

1) Hosting toolbars, menu bars, status bar, and rebars
2) Basic view manipulation, including the resizing of the view in sync with the frame
3) Rebar band manipulation, including adding and resizing of bands
4) Chevron menus[1]

---

1) [1] The newer version of common controls dll provides a way to make tools that have been covered by another band of a rebar control accessible to the user. You can have a chevron to be displayed for toolbars that have been covered. When a user clicks the chevron, a menu is displayed that allows him or her to use the hidden tools.

5) Keyboard accelerators
6) Tool tips for the toolbar buttons
7) Displaying help string in the status bar
8) Displaying frame's icon

In order to create an SDI app, you need to:

1) Derive your frame class from `CFrameWindowImpl`
2) Add the WTL's `DECLARE_FRAME_WND_CLASS` macro, specifying the resource ID of the toolbar and the menu
3) Add the message map and the corresponding message handlers, which should include chaining to the frame base class

For example, you can write our simple SDI example `CMainFrame` class using WTL like so:

```
class CMainFrame :  public CFrameWindowImpl<CMainFrame> {
public:
    DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)
    BEGIN_MSG_MAP(CMainFrame)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
        MESSAGE_HANDLER(WM_CREATE, OnCreate)
        COMMAND_ID_HANDLER(ID_FILE_EXIT, OnFileExit)
        COMMAND_ID_HANDLER(ID_HELP_ABOUT, OnAbout)
        CHAIN_MSG_MAP(CFrameWindowImpl<CMainFrame>)
    END_MSG_MAP()

    LRESULT OnCreate(UINT, WPARAM, LPARAM, BOOL&) {
        // Create the Toolbar and set
        // CFrameWindowImplBase::m_hWndToolBar
        CreateSimpleToolBar();

        // And the statusbar
        CreateSimpleStatusBar();
        return 0;
```

```
    }

    LRESULT OnFileExit(WORD,WORD wID, HWND,BOOL&) {
        SendMessage(WM_CLOSE); // Frame knows to PostQuitMessage
        return 0L;
    }
…
};
```

The `DECLARE_FRAME_WND_CLASS` sets the "common resource id" for the frame. You can associate this resource id with a string in the string table to use as the frame's title, a menu resource, an accelerator table, an icon, and a toolbar resource, all optionally. If WTL finds any resources with the common resource id during creation of the frame window, it loads them automatically. The single exception is the toolbar resource, which you must load manually using the `CreateSimpleToolBar` member function. The status bar needs no associated resource.

The use of `CHAIN_MSG_MAP` in the message map enables routing of messages to the `CFrameWindowImpl`, which, in turn, takes care of handling messages like `WM_SIZE` (to resize the toolbar and status bar) and `WM_DESTROY` (calls `PostQuitMessage`).

To support basic WTL functionality, you must augment `stdafx.h` to include `atlapp.h` and `atlframe.h`:

```
// stdafx.h
#define WIN32_LEAN_AND_MEAN
#include <atlbase.h>
#include <atlapp.h>
extern CAppModule _Module;
#include <atlwin.h>
#include <atlframe.h>
```

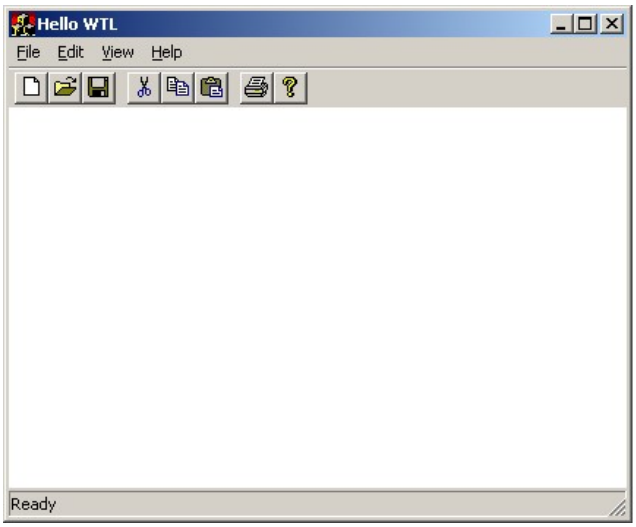
The `atlapp.h` header defines the `CAppModule` structure, which derives from the standard ATL `CComModule` and adds support for a variable number of message loops. The `atlframe.h` header defines the `CFrameWindowImpl` class. To create the SDI window in our WTL world, we need to instantiate the `CMainFrame` object and call its base class member function `CreateEx` in `WinMain`:

```
CAppModule _Module;
int APIENTRY WinMain(…) {
    ::InitCommonControls();
    _Module.Init(NULL, hInstance);

    CMainFrame wndMain;
    wndMain.CreateEx();
    wndMain.ShowWindow(nCmdShow);
    wndMain.UpdateWindow();

    CMessageLoop theLoop;
    _Module.AddMessageLoop(&theLoop);

    int nRet = theLoop.Run();
    _Module.RemoveMessageLoop();

    _Module.Term();
    return nRet;
}
```

The `CreateEx` member function calls the `CWindowImpl::Create` base class member function as well as loading the resources associated with the common resource id. The `CMessageLoop` object replaces our need for a manual message pump. ATL allows one message pump per thread, which is how Multi-SDI is implemented. We will cover the details of `CMessageLoop` and `CAppModule` in Part 2 of this series when we study WTL's message routing architecture.

So far, using WTL, our SDI application looks like **Figure 5a**.

Figure 5a: SDI Application using WTL



We managed to simplify our code and add a new feature: toolbars. Still, toolbars and menu bars are old fashioned. Users want fancy command bars like they see in the Internet Explorer and MS Office. And since it's the users writing the checks…

In Command with Command Bars

A command bar is a tool bar that looks like a Windows menu and has bitmapped menu items. Writing command bars from scratch is a pain. If you are an MFC programmer, Paul DiLascia already did it for you in the January 1998, MSJ (*Give Your Applications the Hot New Interface Look with Cool Menu Buttons*). Likewise, if you are a WTL programmer, the extent of your effort is limited to using WTL's CCommandBarCtrl class in your WM_CREATE handler thusly:

```
LRESULT CMainFrame::OnCreate(UINT, WPARAM, LPARAM, BOOL&)
{
    // m_CmdBar is of type CCommandBarCtrl, defined in AtlCtrlw.h
    HWND hWndCmdBar = m_CmdBar.Create(m_hWnd, rcDefault,
                                      0, ATL_SIMPLE_CMDBAR_PANE_STYLE);

    // Let command bar replace the current menu
    m_CmdBar.AttachMenu(GetMenu());
    m_CmdBar.LoadImages(IDR_MAINFRAME);
    SetMenu(NULL);

    // First create a simple toolbar
    HWND hWndToolBar = CreateSimpleToolBarCtrl(m_hWnd,
        IDR_MAINFRAME,FALSE,ATL_SIMPLE_TOOLBAR_PANE_STYLE);

    // Set m_hWndToolBar member
    CreateSimpleReBar(ATL_SIMPLE_REBAR_NOBORDER_STYLE);

    // Add a band to the rebar represented by m_hWndToolBar
    AddSimpleReBarBand(hWndCmdBar);

    // Add another band to the m_hWndToolBar rebar
    AddSimpleReBarBand(hWndToolBar, NULL, TRUE);

    // Create the usual statusbar
    CreateSimpleStatusBar();
    return 0;
}
```

`CCommandBarCtrl` is the WTL class that encapsulates the command bar functionality and typically, `CMainFrame` contains a member of this type. In the aforementioned snippets of the `WM_CREATE` handler, we first create the call `CCommandBarCtrl::Create`, passing the parent window as the main frame. Then, we attach our menu resource by calling `CCommandBarCtrl::AttachMenu` and call `CCommandBarCtrl::LoadImages`, passing the toolbar resource. The command bar uses the toolbar bitmaps and maps them against the corresponding command ids of the menu items. Because the command bars take charge of the cool menus, we need to get rid of the good old default application menu by calling `SetMenu(NULL)`. Then we create the toolbar control and a rebar control passing our toolbar resource by calling `CreateSimpleToolBarCtrl` and `CreateSimpleRebar`, respectively. Finally, we add the commandbar and the toolbar as two bands of the rebar control by calling `CFrameWindowImplBase::AddSimpleRebarBand`. **Figure 5b** shows the SDI application with command bars. It also shows how the calls we made in the aforementioned snippets map to the UI elements.

Figure 5b: SDI Application with Command Bars

A Frame with a View

It is common practice for MFC programmers to represent the client area of a frame window as a separate child window called a view. The frame is responsible for the "decorations," for example, menu bar and toolbar; while the view is concerned merely with presenting the information that represents the real reason for running the application in the first place. This abstraction is especially handy when combined with splitters or MDI, as discussed later in this article.

A view can be anything that has an `HWND`. Give the `HWND` to the frame by setting the frame's `m_hWndClient` member variable during `WM_CREATE`. For example, to move the bitmap painting functionality of our SDI application to a view, you could define a view class like so:

```
class CBitmapView : public CWindowImpl<CBitmapView> {
public:
    CBitmapView();
    ~CBitmapView();
    BEGIN_MSG_MAP(CBitmapView)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
    END_MSG_MAP()
    LRESULT OnPaint(UINT, WPARAM, LPARAM, BOOL&);
private:
    HBITMAP         m_hBmp;
};
```

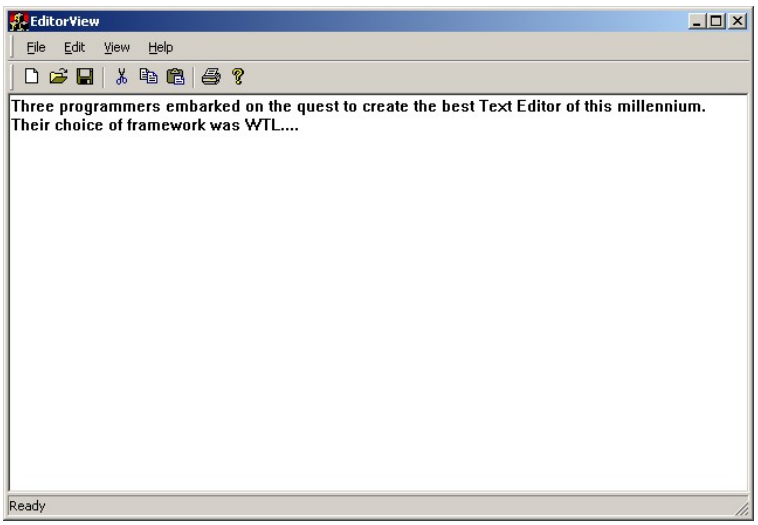
To use this view, do the following:

```
LRESULT OnCreate(UINT, WPARAM, LPARAM, BOOL&) {
…
    // Create the View (m_view is of type CBitmapView)
    m_hWndClient = m_view.Create(m_hWnd, rcDefault, NULL,
                                 WS_CHILD | WS_VISIBLE,
                                 WS_EX_CLIENTEDGE);
    return 0;
}
```

Now, whenever the frame is resized, so is the view, nestled snuggly with the toolbar, command bars, and status bar.

ATL and WTL provide a large range of Window classes that would work fine as views. You could use all of the Windows control and common control wrappers classes as views. For example, by changing the type of the `m_view` class from `CBitmapView` to `CEdit` (as defined in `atlctrls.h`), you have yourself a simple text editor (see **Figure 6**).

Figure 6: Simple Text Editor



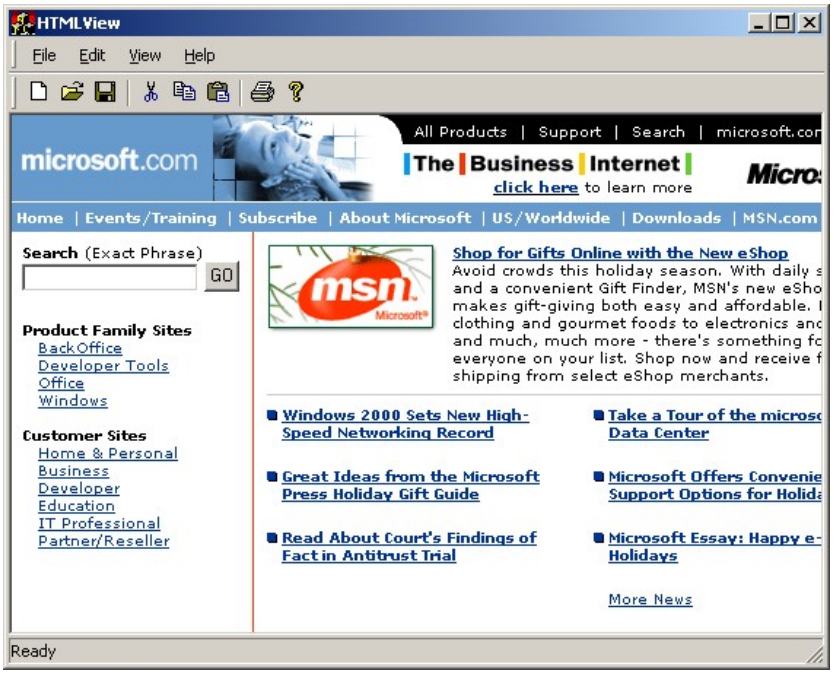
If you want to inherit from one of ATL's window wrapper classes to use as a view, you have to inherit in two ways: the C++ way and the Windows way; that is, super-classing so that you can add functionality and handle messages. For example, by deriving from `CAxWindow`, we could have a custom view that hosts HTML.

```
// Inherit the C++ way to inherit CAxWindow functionality
class CHtmlView : public CWindowImpl<CHtmlView, CAxWindow> {
public:
    // Inherit the Windows way to pre-process Windows messages
    DECLARE_WND_SUPERCLASS(NULL, CAxWindow::GetWndClassName())

    BEGIN_MSG_MAP(CHtmlView)
    …
    END_MSG_MAP()

    CHtmlView() {
        // Init control hosting (defined in atlhost.h)
        AtlAxWinInit();
    }
};
```

We can use our new view to host a web page like so (**Figure 7 displays the result**):

```
LRESULT CMainFrame::OnCreate(UINT, WPARAM, LPARAM, BOOL&)
{
    // Create the View (m_view is of type CHtmlView)
    m_hWndClient = m_view.Create(m_hWnd, rcDefault,
                                 "http://www.microsoft.com",
                                 WS_CHILD | WS_VISIBLE,
                                 WS_EX_CLIENTEDGE);
    return 0;
}
```

Figure 7: Using the New View to Host a Web Page



Dharma Shukla, Chris Sells,
and Nenad Stefanovic

white paper
DEVELOPMENTOR™
www.develop.com

Maintaining the MRU

Although WTL provides support for views, unlike MFC, it does not have a concept of a document. You're on your own if you want to support the document/view pattern in your WTL based apps. To get you started, we wrote a set of classes that give the very basic functionality of a document, which you can use in your apps. See this set of classes in the WTLDocView sample provided with this article.

If you do document work, you're almost certainly going to want to support the Most Recently Used file list in the File menu. Ironically, while WTL doesn't support documents, it does have full support for the MRU via the `CRecentDocumentList` class (defined in `atlmisc.h`). The main frame normally manages an instance of `CRecentDocumentList`. The `WM_CREATE` handler of the main frame initializes `CRecentDocumentList` object with a handle to the File menu, which it can then manage. To populate the MRU, the frame calls the `CRecentDocumentList::ReadFromRegistry` member function to look for up to sixteen entries under a specific Registry key. By default, it sets the limit of MRU documents as four, but you can change that by calling the `CRecentDocumentList::SetMaxEntries` member function.

```
LRESULT CMainFrame::OnCreate(UINT, WPARAM, LPARAM, BOOL&) {
    …
    m_mru.SetMenuHandle(GetMenu().GetSubMenu(0));
    m_mru.ReadFromRegistry(_T("Software\\MyCompany\\MyCoolApp"));
    m_mru.SetMaxEntries(16);
    …
}
```

To add a document to the MRU list for the first time, you call `CRecentDocumentList::AddToList`:

```
LRESULT CMainFrame::OnFileOpen(WORD, WORD, HWND, BOOL&)
{
       if (m_dlgOpen.DoModal(m_hWnd) == IDOK) {
              // Try to open file
              …

              if (bFileOpened) {
                     USES_CONVERSION;
                     m_mru.AddToList(OLE2T(m_dlgOpen.m_bstrName));
              }
       }
       return 0;
}
```

You add the MRU menu items in the range `ID_FILE_MRU_FIRST` to `ID_FILE_MRU_LAST` (defined in `atlres.h`). You can handle the range using the `COMMAND_RANGE_HANDLER` macro. When the user clicks on one of the MRU menu items, the handler can call `CRecentDocumentList::GetFromList` to retrieve the name of the document. After the document opens, the handler should call `CRecentDocumentList::MoveToTop` or `CRecentDocumentList::`RemoveFromList to indicate whether the document opened or not.

```
LRESULT CMainFrame::OnOpenUsingMRU(WORD, WORD wID, HWND, BOOL&) {
```

```
        TCHAR szDocument[MAX_PATH];
        m_mru.GetFromList(wID, szDocument);

        // Try to open the file specified by the menu item
       …

        if (bFileOpened) m_mru.MoveToTop(wID);
        else m_mru.RemoveFromList(wID);

        return 0;
}
```

Finally, when it is time for your app to shut down, you need to call `CRecentDocumentList::WriteToRegistry`, which adds the array of menu tags that from the memory to the Registry.

```
void CMainFrame::OnFinalMessage(HWND /*hWnd*/)
{
        m_mru.WriteToRegistry(_T("Software\\MyCompany\\MyCoolApp"));
        ::PostQuitMessage(0);
}
```

Multi-SDI

Instead of spawning multiple SDI applications, the Internet Explorer uses multiple top-level SDI windows managed by a single application. If you want this feature in your application, you can use WTL's support for Multi-SDI. The design philosophy behind the Multi-SDI application is simple.

- Each top-level window (and its children) runs on a separate thread.
- Each of the threads has a message pump and is thus a UI thread, capable of processing messages and dispatching to the windows it owns. This means not only do all the threads share process-wide data, but also when one thread is busy processing something, the other threads are still fully responsive.
- Thus, each top-level frame window acts as an independent application to the user.

The trick for creating such an application lies in designing a thread manager class, which:

1) Creates a new SDI frame window for each spawned thread
2) Manages the thread handles and the count of the live threads
3) Keeps track of the command line parameters and the initial window size

WTL does not provide any base classes for Multi-SDI apps. However, it generates code for creating such apps with the WTL AppWizard (discussed in full towards the end of this article). The WTL wizard generates a thread manager class that looks like this:

```
class CThreadManager {
public:
	// thread init param
	struct _RunData
	{
		LPTSTR lpstrCmdLine;
		int nCmdShow;
	};

	DWORD m_dwCount;		//count of threads
	HANDLE m_arrThreadHandles[MAXIMUM_WAIT_OBJECTS - 1];

	CThreadManager() : m_dwCount(0) {}
	DWORD AddThread(LPTSTR lpstrCmdLine, int nCmdShow);
	void RemoveThread(DWORD dwIndex);
	int Run(LPTSTR lpstrCmdLine, int nCmdShow);
};
```

`WinMain` will instantiate the thread manager.

```
int WINAPI WinMain(…) {
	hRes = _Module.Init(NULL, hInstance);

	CThreadManager mgr;
	int nRet = mgr.Run(lpstrCmdLine, nCmdShow);

	_Module.Term();
	return nRet;
}
```

The `CThreadManager::Run` method creates a new UI thread and waits on all the UI threads using `MsgWaitForMultipleObjects`. If `MsgWaitForMultipleObjects` indicates that one of the threads has terminated, it decrements its count and returns to waiting. Once all the threads finish, the `Run` method returns and our application shuts down. On the other hand, if `MsgWaitForMultipleObjects` indicates a Windows message, and that message is `WM_USER` (which WTL uses on the thread manager thread to indicate the request of a new UI thread), the thread manager creates a new UI thread.

```
int CThreadManager::Run(LPTSTR lpstrCmdLine, int nCmdShow) {
```

```
        MSG msg;
        // force message queue to be created
        ::PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);

        AddThread(lpstrCmdLine, nCmdShow);

        int nRet = m_dwCount;
        DWORD dwRet;
        while(m_dwCount > 0)
        {
               dwRet = ::MsgWaitForMultipleObjects(m_dwCount,
                    m_arrThreadHandles, FALSE, INFINITE, QS_ALLINPUT);

               if(dwRet == 0xFFFFFFFF)
               {
                    ::MessageBox(NULL, _T("ERROR: Wait for multiple
                    objects failed!!!"), _T("MultiSDI_HTMLView"), MB_OK);
               }
               else if(dwRet >= WAIT_OBJECT_0 && dwRet <= (WAIT_OBJECT_0 +
                   m_dwCount - 1))
               {
                    RemoveThread(dwRet - WAIT_OBJECT_0);
               }
               else if(dwRet == (WAIT_OBJECT_0 + m_dwCount))
               {
                      ::GetMessage(&msg, NULL, 0, 0);
                      if(msg.message == WM_USER)
                             AddThread("", SW_SHOWNORMAL);
                      else
                             ::MessageBeep((UINT)-1);
               }
               else
                      ::MessageBeep((UINT)-1);
        }
        return nRet;
}
```

Notice that the WTL wizard generated code uses raw `WM_USER` message instead of using a special user defined message. Because the generated code uses only the main thread to create UI threads and the thread doesn't create any windows itself, there is no danger of a collision with a custom message with this same value (as would normally be the case).

The `AddThread` routine creates the actual thread with a call to the Win32 CreateThread API passing the address of the thread procedure. This is a static method of the `CThreadManager` class called `RunThread`. `RunThread` creates the SDI frame, calls `::ShowWindow API` with the show command passed to the WinMain, then runs the message loop

```
// CThreadManager::RunThread
static DWORD WINAPI RunThread(LPVOID lpData)
{
	CMessageLoop theLoop;
	_Module.AddMessageLoop(&theLoop);

	_RunData* pData = (_RunData*)lpData;

	CMainFrame wndFrame;
	if(wndFrame.CreateEx() == NULL)
	{
		ATLTRACE(_T("Frame window creation failed!\n"));
		return 0;
	}

	wndFrame.ShowWindow(pData->nCmdShow);
	::SetForegroundWindow(wndFrame);  // Win95 needs this
	delete pData;

	int nRet = theLoop.Run();
	_Module.RemoveMessageLoop();
	return nRet;
}
```

The handler for the File/New Window menu item in a Multi-SDI application should post a `WM_USER` message to the main thread to signal the thread manager to create a new thread with a new frame.

```
LRESULT CMainFrame::OnFileNewWindow(WORD, WORD, HWND, BOOL&) {
	::PostThreadMessage(_Module.m_dwMainThreadID, WM_USER, 0, 0L);
	return 0;
}
```

Figure 8a & 8b shows the Multi-SDI application.

Figure 8a: Multi-SDI Application

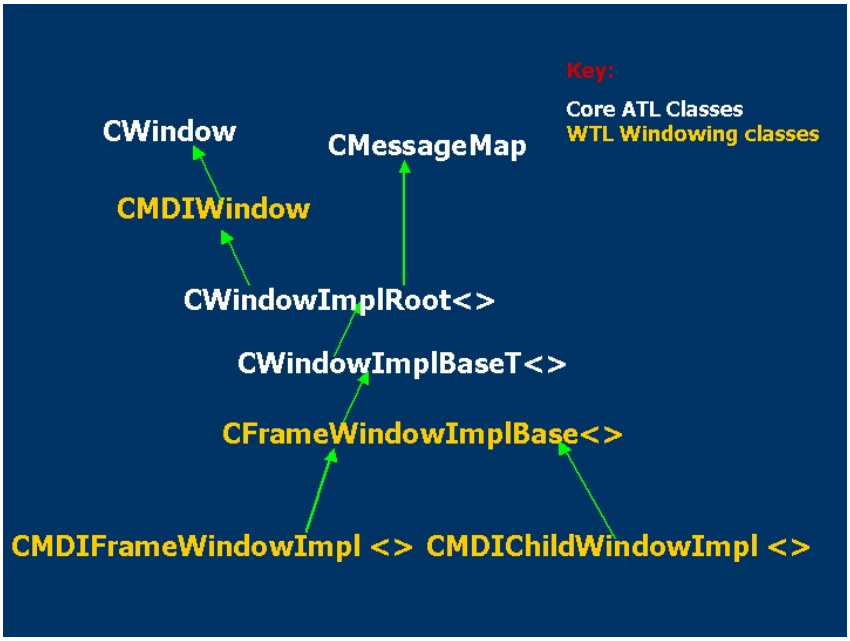
Figure 8b: Multi-SDI Application

Multi-SDI applications lend themselves to single instance activation, that is, when the user double-clicks on YourMultiSdi.exe, you would probably prefer a new thread to a new instance of the application. WTL does not support this, but you can add it fairly easily using DDE. For an example, see the SingleInstance sample that accompanies this article for an example.

Creating MDI Apps

Before Multi-SDI, there was MDI. WTL provides a set of classes based on the `CFramewindowImplBase` that enables us to create MDI apps. **Figure 9** shows the MDI windowing hierarchy of WTL.

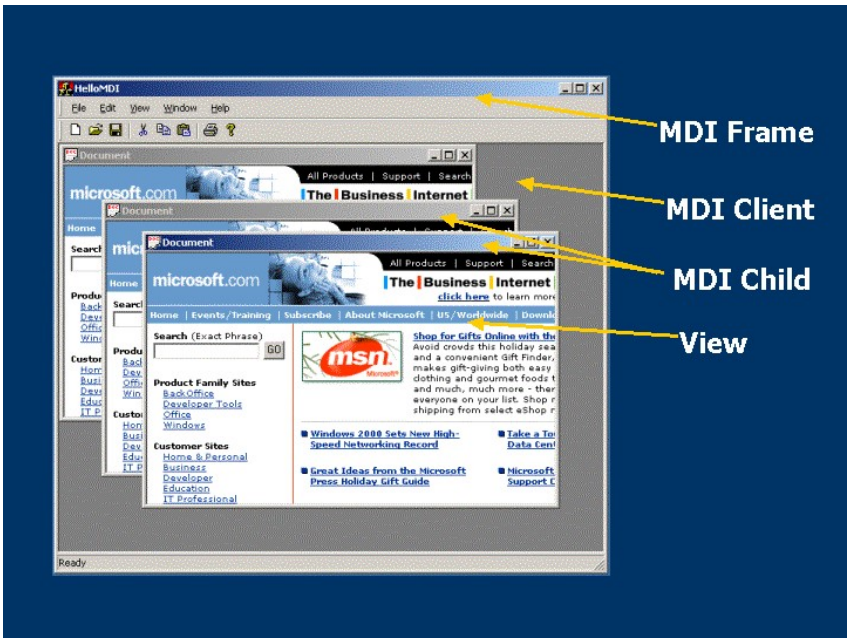
Figure 9: MDI Windowing Hierarchy of WTL



`CMDIWindow` inherits from `CWindow` and stores the `HWND` to the MDI client window and the MDI Frame menu, as well as providing simple wrappers that send the `WM_MDIXXX` family of messages.

MDI applications consist of the frame, the client area of the frame that manages the MDI children, and the MDI children themselves (as shown in **Figure 10**).

Figure 10: MDI Application Consisting of Frame, Client Area, and MDI Children



To create an MDI child window, you need to derive your class from WTL's `CMDIChildWindowImpl`.

```
class CChildFrame : public CMDIChildWindowImpl<CChildFrame> {
public:
	DECLARE_FRAME_WND_CLASS(NULL, IDR_MDICHILD)

	CHtmlView m_view;

	virtual void OnFinalMessage(HWND /*hWnd*/) {
		delete this;
	}

	BEGIN_MSG_MAP(CChildFrame)
		MESSAGE_HANDLER(WM_CREATE, OnCreate)
		CHAIN_MSG_MAP(CMDIChildWindowImpl<CChildFrame>)
	END_MSG_MAP()

	LRESULT OnCreate(UINT,WPARAM,LPARAM,BOOL&) {
		m_hWndClient = m_view.Create(m_hWnd, rcDefault,
				_T("http://www.microsoft.com"),
				WS_CHILD | WS_VISIBLE, WS_EX_CLIENTEDGE);

		bHandled = FALSE; // Let base class have a crack
		return 1;
	}
	…
};
```

An MDI child frame window is very similar to an SDI frame. For example, we pass the menu resource id of the MDI child to the `DECLARE_FRAME_WND_CLASS` macro to take care of associated resource creation, just like in our SDI frame. Also, we handle `WM_CREATE` to create our view. However, notice that after creating our view, we set `bHandled` to `FALSE` to indicate that we want the MDI child base class to handle the `WM_CREATE` message. Finally, notice that our `OnFinalMessage` handler calls `delete` on itself. This allows the child to reclaim its own resources, freeing the frame from this responsibility.

Working our way out, the `CMDIFrameWindowImpl` implements both the MDI frame and MDI client, which will serve as the base class for your MDI frame, for example,

```
class CMainFrame : public CMDIFrameWindowImpl<CMainFrame> {
public:
	DECLARE_FRAME_WND_CLASS(NULL, IDR_MAINFRAME)
	BEGIN_MSG_MAP(CMainFrame)
		MESSAGE_HANDLER(WM_CREATE, OnCreate)
		COMMAND_ID_HANDLER(ID_FILE_NEW, OnFileNew)
		COMMAND_ID_HANDLER(ID_WINDOW_CASCADE, OnWindowCascade)
		…
		CHAIN_MSG_MAP(CMDIFrameWindowImpl<CMainFrame>)
	END_MSG_MAP()

	LRESULT OnCreate(UINT, WPARAM, LPARAM, BOOL&)
	{
```

```
            // Create command bar window, toolbar and statusbar
            …

            // Create MDI client
            CreateMDIClient();
            m_CmdBar.SetMDIClient(m_hWndMDIClient);
            …
            return 0;
        }

        LRESULT OnFileNew(WORD, WORD, HWND, BOOL&)
        {
             // Create MDI child
            CChildFrame* pChild = new CChildFrame;
            pChild->CreateEx(m_hWndMDIClient);
            return 0;
        }

        LRESULT OnWindowCascade(WORD,WORD,HWND, BOOL&)
        {
            MDICascade();
            return 0;
        }
…
};
```

Our WM_CREATE handler calls the `CMDIFrameWindowImpl` base-class member function `CreateMDIClient`, which in-turn creates the MDI client and sets `m_hWndMDIClient`. The MDI client window parents the MDI child frame windows (containing the HTML View, in our case). Each new MDI child is created in the `OnFileNew` handler, which creates a new instance of our MDI child window class, using the MDI client as the parent. Finally, we create the MDI Frame window in the `WinMain` in the same way we did in the SDI case.

Divide and Conquer with Splitters

Both Multi-SDI and MDI is about a single application showing multiple views on an application's data. Another popular method is to split a window into one or more child windows using a splitter control. WTL provides the support for splitters with the `CSplitterImpl`, `CSplitterWindowImpl`, and `CSplitterWindow` classes. The core of WTL's splitter architecture, the `CSplitterImpl` class, provides all the necessary magic to create the splitter window, to resize the panes, and so on. `CSplitterImpl` it is designed to work as a base class. The class that derives from `CSplitterImpl` should be a `CWindowImpl` derived class (or is capable of processing messages by deriving from `CMessageMap`). Even though `CSplitterImpl` has its own message map, it depends on its deriving class to chain messages to it.

```
//atlsplit.h
template <class T, bool t_bVertical = true>
class CSplitterImpl
{…

       BEGIN_MSG_MAP(thisClass)
              MESSAGE_HANDLER(WM_CREATE, OnCreate)
              MESSAGE_HANDLER(WM_PAINT, OnPaint)
              MESSAGE_HANDLER(WM_PRINTCLIENT, OnPaint)
              if(IsInteractive())
              {
                     MESSAGE_HANDLER(WM_SETCURSOR, OnSetCursor)
                     MESSAGE_HANDLER(WM_MOUSEMOVE, OnMouseMove)
                     MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
                     MESSAGE_HANDLER(WM_LBUTTONUP, OnLButtonUp)
                     MESSAGE_HANDLER(WM_LBUTTONDBLCLK,
                                          OnLButtonDoubleClick)
              }
              MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
              MESSAGE_HANDLER(WM_MOUSEACTIVATE, OnMouseActivate)
              MESSAGE_HANDLER(WM_SETTINGCHANGE, OnSettingChange)
       END_MSG_MAP()
…
…
};
```

The following code shows how to convert the SDI Frame window that we earlier developed into a frame with a splitterbar dividing two HTML views.

```
class CMainFrame :
    public CFrameWindowImpl<CMainFrame>,
    public CSplitterImpl<CMainFrame,true>
{
public:
       DECLARE_WND_CLASS_EX(NULL, CS_DBLCLKS, COLOR_WINDOW)

       typedef CFrameWindowImpl<CMainFrame> winbaseClass;
       typedef CSplitterImpl< CMainFrame,true> splitbaseClass;
       BEGIN_MSG_MAP(CMainFrame)
              …
              MESSAGE_HANDLER(WM_CREATE, OnCreate)
              MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBackground)
              MESSAGE_HANDLER(WM_SIZE, OnSize)
              CHAIN_MSG_MAP(splitbaseClass)
       END_MSG_MAP()
       //…
private: //the two panes within the splitter
       CHtmlView           m_ViewLeft;
       CHtmlView           m_ViewRight;
```

```
        …
};
```

The `WM_CREATE` handler of the `CMainFrame` class creates the two panes and splits them by calling the `CSplitterImpl::SetSplitterPanes` method. Finally, it sets the initial splitter positions by calling `CSplitterImpl::SetSplitterPos`.

```
LRESULT CMainFrame::OnCreate(UINT, WPARAM,LPARAM, BOOL& bHandled)
{
        …

        // First the 2 views (panes)
        m_ViewLeft.Create(m_hWnd, rcDefault,
             _T("http://www.sellsbrothers.com/tools"), WS_CHILD |
             WS_VISIBLE | WS_CLIPSIBLINGS,
             WS_EX_STATICEDGE);
        m_ViewRight.Create(m_hWnd, rcDefault,
             _T("http://www.sellsbrothers.com/comfun"), WS_CHILD |
             WS_VISIBLE | WS_CLIPSIBLINGS|WS_VSCROLL | WS_CLIPCHILDREN,
             WS_EX_STATICEDGE);

        // Link the panes to the splitter
        SetSplitterPanes(m_ViewLeft, m_ViewRight);

        // Set the initial positions
        RECT rc;GetClientRect(&rc);
        SetSplitterPos((rc.right - rc.left) / 4);

        bHandled = FALSE;
        return 0;
}
```

The `WM_SIZE` handler calculates the available client area minus the area occupied by the Rebar control and calls `CSplitterImpl::SetSplitterRect` to reposition the splitter. Notice we don't do anything when the application minimizes the window.

```
LRESULT CMainFrame::OnSize(UINT, WPARAM wParam, LPARAM, BOOL& bHandled)
{
	if(wParam != SIZE_MINIMIZED)
	{
		RECT rc; GetClientRect(&rc);
		RECT rcBar; ::GetClientRect(m_hWndToolBar,&rcBar);
		rc.top = rcBar.bottom;
		SetSplitterRect(&rc);
	}

	bHandled = FALSE;
	return 1;
}
```

Finally, to eliminate the flicker during resizing, `CMainFrame` handles `WM_ERASEBKGND` and bypasses the default action done by `DefWindowProc`. The `DefWindowProc` function erases the background by using the window class' background brush, which can cause unwanted flicker. Processing this message and returning a non-zero value indicates that no further erasing is required.

```
LRESULT CMainFrame::OnEraseBackground(UINT, WPARAM, LPARAM, BOOL&)
{
	return 1;
}
```

Figure 11a shows we mean to use the app. `CSplitterImpl` in the inheritance scenario; that is, the window class that wants to split its client area into two panes has to derive from `CSplitterImpl`.

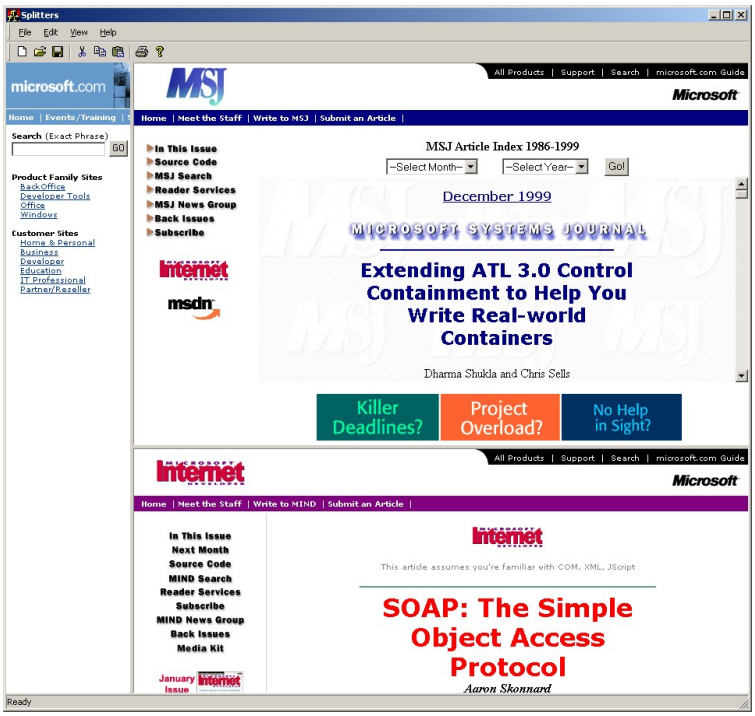
Figure 11a: `CSplitterImpl` in the Inheritance Scenario

To handle the `WM_ERASEBKGND` and `WM_SIZE` messages every time by deriving from `CSplitterImpl` could be painful. Again, WTL comes to the rescue by providing another layer of abstraction in between the `CSplitterImpl` and your class. WLT does this by using `CSplitterWindowImpl`, which takes care of handling `WM_ERASEBKGND` and `WM_SIZE` for you. `CSplitterWindowImpl` also takes care of chaining the messages to `CSplitterImpl` class and forwarding the notifications (`WM_NOTIFY`, `WM_COMMAND`, `WM_*SCROLL` etc.) to the parent windows using `FORWARD_NOTIFICATIONS` macro.

```
//atlsplit.h
template <class T, bool t_bVertical = true, class TBase = CWindow, class
TWinTraits = CControlWinTraits>
class ATL_NO_VTABLE CSplitterWindowImpl :
      public CWindowImpl< T, TBase, TWinTraits >,
       public CSplitterImpl< CSplitterWindowImpl<T ,
                      t_bVertical, TBase, TWinTraits >,
                      t_bVertical>
{
public:
       DECLARE_WND_CLASS_EX(NULL, CS_DBLCLKS, COLOR_WINDOW)

       typedef CSplitterWindowImpl< T , t_bVertical, TBase, TWinTraits >
             thisClass;
       typedef CSplitterImpl<
       CSplitterWindowImpl<T,t_bVertical,TBase, TWinTraits >,
            t_bVertical>baseClass;

       BEGIN_MSG_MAP(thisClass)
             MESSAGE_HANDLER(WM_ERASEBKGND, OnEraseBackground)
             MESSAGE_HANDLER(WM_SIZE, OnSize)
             CHAIN_MSG_MAP(baseClass)
             FORWARD_NOTIFICATIONS()
       END_MSG_MAP()

       LRESULT OnEraseBackground(UINT,WPARAM, LPARAM, BOOL&){return 1;}
       LRESULT OnSize(UINT, WPARAM wParam, LPARAM, BOOL& bHandled)
       {
             if(wParam != SIZE_MINIMIZED)
                   SetSplitterRect();
             bHandled = FALSE;
             return 1;
       }
};
```

You can mix-and-match the splitter support of WTL to divide your client area in both the axes into nested multiple views. The NestedSplitters sample application that accompanies this article demonstrates that. (see **Figure 11b**).

Figure 11b: NestedSplitters Sample Application



If you prefer to contain your splitter window controls instead of inherit from them, WTL provides `CSplitterWindow` and `CHorSplitterWindow`. The following provides an example of `CSplitterWindow` used in a windowed COM control created with the good old ATL COM AppWizard.

```
class ATL_NO_VTABLE CFooControl :
       public CComObjectRootEx<CComSingleThreadModel>,
       public CComControl<CFooControl>,…
{
       CFooControl()
       {
              m_bWindowOnly = TRUE;
       }

       BEGIN_MSG_MAP(CFooControl)
              MESSAGE_HANDLER(WM_CREATE, OnCreate)
              CHAIN_MSG_MAP(CComControl<CFooControl>)
              DEFAULT_REFLECTION_HANDLER()
       END_MSG_MAP()

       // Message handlers and other usual suspects

private:
       CSplitterWindow     m_splitter;
       CHtmlView           m_LeftView;
       CHtmlView           m_RightView;
};
```

Once again the `WM_CREATE` handler plays its part.

```
LRESULT CFooControl::OnCreate(UINT, WPARAM, LPARAM, BOOL&)
{
       AtlAxWinInit();

       //Create the Left and right views
       m_LeftView.Create(m_hWnd, rcDefault,
       _T("http://www.microsoft.com"),WS_CHILD | WS_VISIBLE |
      WS_CLIPSIBLINGS | WS_CLIPCHILDREN,WS_EX_STATICEDGE);

       m_RightView.Create(m_hWnd, rcDefault,
       _T("http://www.microsoft.com"),WS_CHILD | WS_VISIBLE |
      WS_CLIPSIBLINGS | WS_CLIPCHILDREN,WS_EX_STATICEDGE);
       RECT rect; GetClientRect(&rect);

       //Create the splitter object
       m_splitter.Create(m_hWnd, rect, NULL, WS_CHILD | WS_VISIBLE |
       WS_CLIPSIBLINGS | WS_CLIPCHILDREN);

       m_splitter.SetSplitterPanes(m_LeftView, m_RightView);
       m_splitter.SetSplitterPos((rect.right - rect.left) / 4);

       return 0L;
}
```

Finally, the WTL splitter classes also offer the functionality of dynamically flipping different views within a pane by calling `CSplitterImpl::SetSplitterPane` with a different `hWnd`. It also allows you to restrict the splitter positions by setting `CSplitterImpl::m_cxyMin` and calling `CSplitterImpl::SetSplitterPos`.

GDI Wrappers

No matter how many views you have or how you arrange them, somebody has to do the drawing in them. WTL provides simple wrappers around all of the Win32 GDI objects including those in **Table 2**.

Table 2: WTL Wrappers for Win32 GDI Objects

HDC	CDCT
HPEN	CPenT
HBRUSH	CBrushT
HFONT	CFontT
HBITMAP	CBitmapT
HRGN	CRgnT
HPALETTE	CPalletT

In fact, there are two versions of wrappers for each of the GDI resource it wraps: managed and unmanaged. The managed version destroys the object that it holds, whereas the unmanaged leaves the destruction of the underlying GDI object to the client of the object. For example, the `CFontT` class is a wrapper around the `HFONT` but it takes a Boolean template argument specifying whether it is a *managed object* or an *unmanaged handle*. The idea is to provide the flexibility to the client code that uses these wrappers.

```
//atlgdi.h
typedef CFontT<false>            CFontHandle;
typedef CFontT<true>       CFont;

template <bool t_bManaged>
class CFontT
{
public:
// Data members
       HFONT m_hFont;

// Constructor/destructor/operators
       CFontT(HFONT hFont = NULL) : m_hFont(hFont)
       { }

       ~CFontT()
       {
```

```
            if(t_bManaged && m_hFont != NULL)
                  DeleteObject();
      }

      CFontT<t_bManaged>& operator=(HFONT hFont)
      {
            m_hFont = hFont;
            return *this;
      }
…
};
```

Notice the `CFontT` destructor checks for the `t_bManaged` template argument for *true* before calling the `DeleteObject` method. The two versions of this class are defined for direct use, and the resulting types are `CFont` and `CFontHandle`, which are the managed and the unmanaged versions, respectively. WTL takes the exact same approach for all of the GDI wrappers.

The undisputed wrapper champion in all of WTL is `CDCT`, which wraps over 240 methods covering the GDI and the WGL[2] functions. Four other WTL classes derive from the managed version of this class (`CDC`): `CPaintDC`, `CWindowDC`, `CclientDC`, and `CEnhMetaFileDC`. The `CPaintDC` class wraps the windows with `PAINTSTRUCT` and uses the `BeginPaint/EndPaint` pair to obtain/release the DC. The `CClientDC` class represents a Windows client and uses `GetDC` and `ReleaseDC` to obtain/release the DC. The `CWindowDC` class uses `GetWindowDC` and `ReleaseDC` to obtain and destroy the DC for the entire window. Finally, `CEnhMetaFileDC` wraps an `HENHMETAFILE` and calls `CreateEnhMetaFile` and `DeleteEnhMetaFile` to create and destroy the Windows meta file, respectively. The following `CPaintDC` declaration shows the way this family of classes works.

```
//atlgdi.h
class CPaintDC : public CDC
{
public:
      HWND m_hWnd;
      PAINTSTRUCT m_ps;

      CPaintDC(HWND hWnd)
      {
            m_hWnd = hWnd;
            m_hDC = ::BeginPaint(hWnd, &m_ps);
      }

      ~CPaintDC()
      {
            ::EndPaint(m_hWnd, &m_ps);
```

[2] WGL API consists of a set of "wiggle" functions (named after the wgl prefix to its name). A wiggle function is a Win32 function that bridges the gap between the portable OpenGL™ graphics library and the Microsoft Windows® platform.

```
            Detach();
        }
};
```

Using the managed GDI wrappers can simplify raw GDI code considerably. For example, our bitmap view can be reduced to the following:

```
class CBitmapView : public CWindowImpl<CBitmapView>
{
public:
    CBitmapView()
    {
        m_bmp.LoadBitmap(MAKEINTRESOURCE(IDB_ATLWINDOWING));
    }

    BEGIN_MSG_MAP(CBitmapView)
        MESSAGE_HANDLER(WM_PAINT, OnPaint)
    END_MSG_MAP()

    LRESULT OnPaint(UINT, WPARAM, LPARAM, BOOL&)
    {
        CPaintDC    dc(m_hWnd);
        RECT        rect; GetClientRect(&rect);
        CDC         dcMem; dcMem.CreateCompatibleDC(dc);

        CBitmap     bmpOld = dcMem.SelectBitmap(m_bmp);
        BITMAP      bm; m_bmp.GetBitmap(&bm);
        SIZE        size = { bm.bmWidth, bm.bmHeight };

        dc.BitBlt(rect.left, rect.top, size.cx, size.cy,  dcMem,
                  0, 0, SRCCOPY);

        // Cleanup
        dcMem.SelectBitmap(bmpOld);

        return 0;
    }

private:
    CBitmap m_bmp;
};
```

To further aid the regular GDI needs of your app, WTL provides a set of inline global helpers all starting with the Atl prefix. These include `AtlGetStockBrush`, `AtlGetStockFont`, `AtlGetStockPalette`, `AtlGetStockPen`, `AtlLoadAccelerators`, `AtlLoadBitmap`, `AtlLoadBitmapImage`, `AtlLoadCursor`, and `AtlLoadString` to name a few.

Dharma Shukla, Chris Sells,
and Nenad Stefanovic

white paper
DEVELOPMENTOR™
www.develop.com

CString, et al.

There's a little box on the first screen of the ATL COM AppWizard called "Support MFC." Because ATL supports basic windowing and dialogs, which is all that is likely to be needed in most COM servers, why would anyone want to check this box? One reason: `CString`. There are many developers that depend on the subtleties of `CString` and they are willing to pay the space overhead of MFC in an ATL server just to have it. Now they don't have to. The ATL team has ported MFC's `CString` to ATL and has shipped it with the rest of WTL (defined in `atlmisc.h`). The goal is transparent compatibility. The following steps show how the two implementations compare:

1) WTL's `CString` is copy-on-write, just like MFC's.
2) Many of the WTL methods have versions that take `CString` parameters, just like MFC.
3) WTL's `CString` has all the methods that are in the MFC counterpart except overloaded versions of a few of methods (discussed next) and `CollateNoCase`, an NLS-aware comparison.
4) WTL's `TrimRight` and `TrimLeft` methods just strip out the white space, whereas MFC's versions strip out any character/or a set of characters you may specify.
5) WTL's `Find` method starts from the left looking for a particular character in the string. MFC provides two overloaded versions of `Find`. One is identical to WTL's and the other allows you to pass a zero based index marking your search for the character in the string.
6) Unfortunately, like MFC, WTL's `CString` uses the CRT, making its usage in CRT-less ATL COM servers a problem.
7) WTL's version of `AllocBuffer` and `AllocBeforeWrite` methods return `BOOL` unlike MFC's, which return a `void`.
8) WTL's version of `Format` does not support floating point whereas MFC's does.

In addition to `CString`, WTL provides other useful MFC-like wrappers, including `CRect`, `CSize`, `CPoint`, `CFileFind` (which wraps the `WIN32_FIND_DATA` structure), and `CWaitCursor` (for displaying the ever popular hour glass during a long operator). Notably missing from WTL's arsenal of frequently used helper classes are the wrappers for `CURRENCY` and `DATE`, which MFC has support for in terms of `COleCurrency` and `COleDateTime/COleDateTimeSpan`. You can download classes that perform these functions from the web site http://www.sellsbrothers.com/tools.

WTL's support for DDX

Another of MFC's features that WTL copies is Dynamic Data Exchange (DDX). DDX is the act of moving data back and forth between a Window object's data members and a window's child controls. WTL provides support for DDX via the `CWinDataExchange` class and a set of macros that implement the `DoDataExhange` method of this class.

```
// atlddx.h
template <class T> class CWinDataExchange {
public:
       // Data exchange method - override in your derived class
       BOOL DoDataExchange(BOOL /*bSaveAndValidate*/ = FALSE, UINT
```

```
            /*nCtlID*/ = (UINT)-1)
      {
            // this one should never be called, override it in
            // your derived class by implementing DDX map
            ATLASSERT(FALSE);
            return FALSE;
      }
      …
};
```

Any class that wants to participate in the DDX ritual simply derives from `CWinDataExchange` and provides an implementation of the `DoDataExchange` method. This method is implemented most easily via the `DDX_MAP` with each entry corresponding to the child controls placed on its client area, for example,

```
class CStringDialog :
    public CDialogImpl<CStringDialog>,
    public CWinDataExchange<CStringDialog>
{
public:
    CStringDialog() { *m_sz = 0; }


    enum { IDD = IDD_STRING };
    BEGIN_MSG_MAP(CStringDialog)
        COMMAND_ID_HANDLER(IDOK, OnOK)
        COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    END_MSG_MAP()

    BEGIN_DDX_MAP(CMainDlg)
        DDX_TEXT(IDC_STRING, m_sz)
    END_DDX_MAP()

      LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&) {
            DoDataExchange(FALSE);  // Populate the controls
            return 0;
      }

    LRESULT OnOK(WORD, WORD wID, HWND, BOOL&) {
        DoDataExchange(TRUE);  // Populate the data members
        EndDialog(wID);
        return 0L;
    }

    LRESULT OnCancel(WORD, WORD wID, HWND, BOOL&) {
        EndDialog(wID);
        return 0L;
    }
```

```
public:
    enum { MAX_STRING = 128 };
    char    m_sz[MAX_STRING+1];
};
```

`CStringDialog` is a dialog that derives from `CWinDataExhange` and contains a single edit control. The dialog first populates the edit control with whatever is in the `m_sz` data member during `WM_INITDIALOG` when the handler calls `DoDataExchange(FALSE)`. The dialog synchronizes the data member to the state of the edit control when the user presses the OK button and the handler calls `DoDataExchange(TRUE)`. This is very similar to MFC's DDX implementation and the usage.

```
LRESULT CMainFrame::OnFileTitle(WORD,WORD wID, HWND,BOOL&) {
    CStringDialog   dlg;
    GetWindowText(dlg.m_sz, dlg.MAX_STRING);
    if( dlg.DoModal() == IDOK ) SetWindowText(dlg.m_sz);
    return 0L;
}
```

In addition to text, WTL's DDX supports signed and unsigned integers and floating-point numbers. It also supports controls like radio buttons and checkboxes. **Table 3** shows the macros that `atlddx.h` defines.

Table 3: WTL's DDX Macros

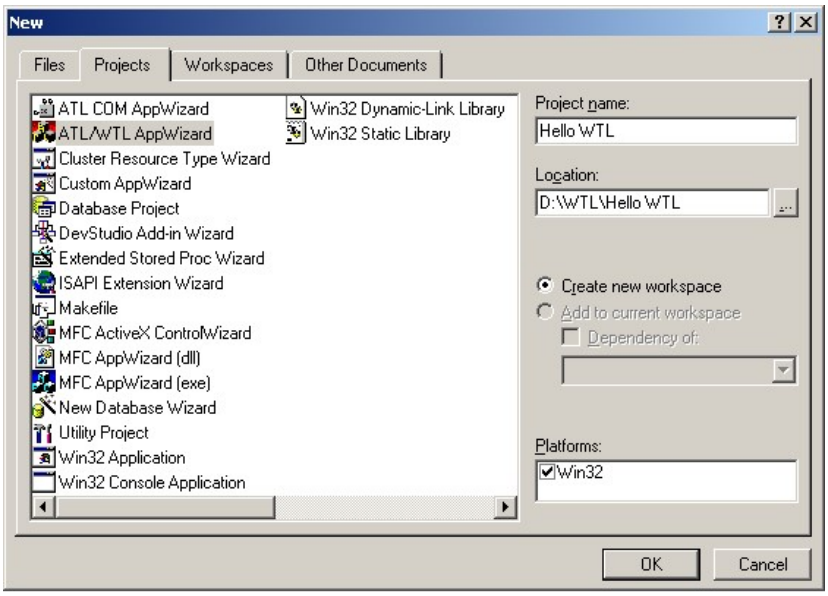
Macro	Purpose
`DDX_TEXT(nID, var)`	Associates the text content of a control with the `CString`, `CComBSTR` or `LPTSTR` member variable of your class.
`DDX_TEXT_LEN(nID, var, len)`	Same as `DDX_TEXT` but also validates the length.
`DDX_INT(nID, var)`	Associates the numeric value that the user typed in a control with the integer class member.
`DDX_INT_RANGE(nID, var, min, max)`	Same as `DDX_INT` but also validates the range.
`DDX_UINT(nID, var)`	Same as `DDX_INT` for unsigned Integers.
`DDX_UINT_RANGE(nID, var, min, max)`	Same as `DDX_INT_RANGE` for unsigned integers.
`DDX_FLOAT(nID, var)`	Same as `DDX_INT` for floats.
`DDX_FLOAT_RANGE(nID, var, min, max)`	Same as `DDX_INT_RANGE` for floats.
`DDX_CONTROL(nID, obj)`	`DDX_CONTROL` subclasses a control, represented by the *obj* parameter in the macro (just like in MFC). Because of that, your data member must derive from `CWindowImpl` (or at least have `SubclassWindow` method).
`DDX_CHECK(nID, var)`	Sets the var to the checked state of the button.
`DDX_RADIO(nID, var)`	Manages the transfer of integer data between a radio control group and a int data member.

Compared to MFC, WTL's DDX doesn't happen automatically; you have to call manually call `DoDataExchange`. This gives you flexibility to call it exactly when you want. Also, you can call it for only one control by passing in the control's id to `DoDataExchange`.

WTL Wizardry

At the beginning of this article we mentioned a VC AppWizard that generates WTL-based applications by way of the MFC AppWizard. To use it, you need copy the `AtlApp60.awx` file from the `WTL\AppWiz` folder to the `C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\Template` folder. This puts the ATL/WTL AppWizard in VC's New Project dialog (as shown in **Figure 12**).

Figure 12: ATL/WTL AppWizard in VC's New Project Dialog



By using the WTL wizard you can create four different types of Applications: SDI, MDI, Multi SDI, and Dialog-based. The control hosting option uses the same functionality provide in ATL 3.0's `atlhost.h`. Choosing to have your application act as a COM server (as shown in **Figure 13**) causes the wizard to generate code very similar to what the ATL COM AppWizard generates if you chose the EXE server option.

Figure 13: Application Acting as a COM Server

Just like MFC, you can choose various frame decoration options like toolbar, commandbar, rebar, and status bar. Choosing these options has an effect on the main application frame window class's `WM_CREATE` handler, which creates these decorations based on your selections. You even can specify a view contained within the application frame. In addition to acting like a Form view or a generic window, you can base the view on listbox, edit, list view, tree view, rich edit based controls, or even an HTML view by using of ATL's control hosting support (as shown in **Figure 14**).

Figure 14: Example of Ways to Base Your View

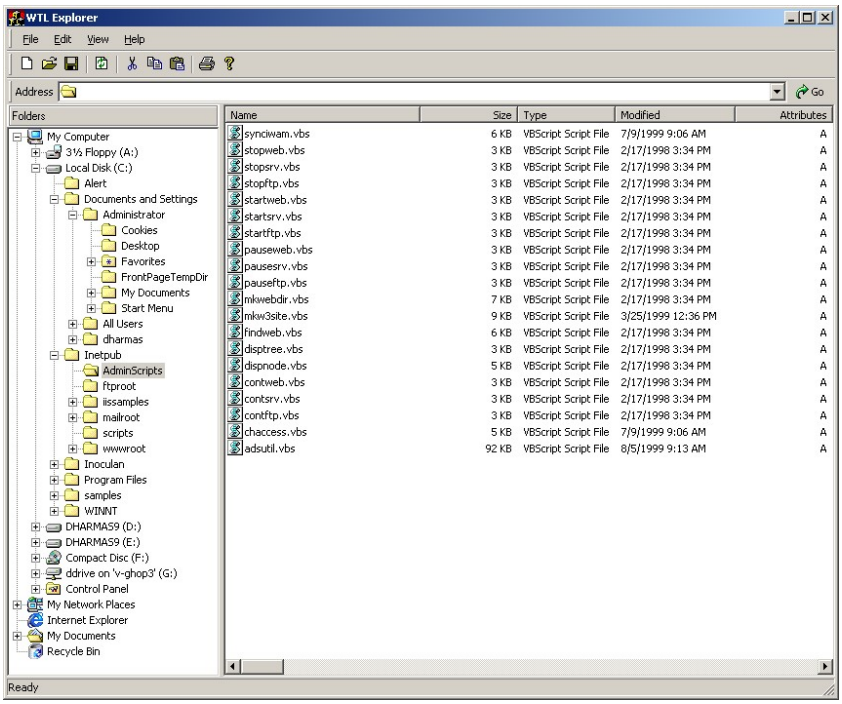
Of course, like all wizards, this wizard does not reflect all capabilities as options, but it reflects enough to give you a good head start on building your WTL-based applications.

WTL Samples

To further your understanding of WTL, WTL ships with three sample apps named GuidGen, MTPad, and MdiDocView. GuidGen, which demonstrates a simple dialog-based application, looks and works exactly like your favorite guidgen.exe (and is smaller). The MTPAd is a Multi-SDI app and demonstrates a lot of WTL's support for advanced functionality (which we discuss in Part 2 of this series), like Common Dialogs, the common control wrappers, UI updating, etc. Finally, MDIDoc view shows the use of WTL's MDI classes.

To get a better idea of writing WTL based apps, you also can check out the samples that come with this article. In fact, we decided to put all our understanding of WTL into a larger sample application that mimicked an old favorite: the Windows File Explorer (**Figure 15**).

Figure 15: Windows File Explorer



Summary

We are half way there. We just covered the WTL's support for SDI, Multi SDI, MDI apps, explorer/workspace style apps that use splitters, GDI wrappers, helper classes, and DDX. We haven't seen WTL command bar architecture, common controls wrappers, message routing architecture including message cracking and filtering, and idle handling yet. Neither have we covered the common dialog wrappers, property pages and sheets, printing support, nor scrolling windows. We'll cover these topics in Part 2 of this series.

Chris Sells is the Director of Software Engineering at DevelopMentor and the co-author of *Effective COM* and *ATL Internals;* he can be reached http://staff.develop.com/csells.

Dharma Shukla is a Software Design Engineer on the BizTalk Server 2000 group at Microsoft and can be reached at dharmas@microsoft.com.

Nenad Stefanovic is a member of the original ATL/WTL team. He is a Software Development Engineer at Microsoft and can be reached at nenads@microsoft.com.

white paper

DEVELOPMENTOR℠

WTL Makes UI Programming a Joy
Part 2: The Bells and Whistles

Recall from Part 1 of this series, that the Windows Template Library (WTL) is available on the January 2000 Platform SDK and beyond. It's an SDK sample produced by members of the ATL team chartered with building an ATL-based wrapper around the windowing portions of the Win32 API. In Part I, I covered WTL's support for SDI, Multi-SDI, MDI, and explorer/workspace-style applications that use command bars, splitters, GDI wrappers, helper classes, and DDX. In Part II, I'll give you a look into WTL's common controls wrappers, common dialog wrappers, property sheets, printing, print preview, scrolling support, message cracking, filtering, and idle handling. I will demonstrate these features while building a new sample program: BmpView.

The BmpView Sample

This sample, called BmpView, displays .BMP files. I created the starting code for BmpView using the Visual C++ WTL App Wizard. I used the default settings—an SDI application type with a generic view window and a rebar, a command bar, and a status bar.

Before I start adding features, I need to remove some. Because BmpView should be able to open BMP files, but doesn't save them (WTL is not PhotoDraw, after all), I need to remove menu items for saving files. I also need menu items for printing, page setup, and print preview. All these changes require me to modify resources of the application, adding or removing menu items and toolbar buttons. The modified application is shown in **Figure 1**.

Figure 1: The Modified Application

Instead of the built-in Win32 menu bars, the wizard-generated code uses a WTL command bar for that polished, modern look. Command bars look like a specialized toolbar, but with menu items instead of the toolbar buttons. Command bars also provide images for drop-down menu items, which assist in recognizing commands by providing a visual link between a text label and a corresponding toolbar button. Part I showed how command bars work. In BmpView, I'd like to apply the same nifty graphics to a context menu, which is what I'll tackle first.

Context Menus

A context menu should appear when the user right-clicks in the main window. Assuming the context menu will be created from a resource in your application, it is pretty straightforward how to handle the `WM_CONTEXTMENU` message (which is sent when the user right-clicks in the main window) on the main window and display the menu using `TrackPopupMenu` windows API. Unfortunately, like the built-in menu bars, Win32 context menus don't show images. Luckily, the same class that I use to show the menu command bar can also be used to show images on a context menu. The command-bar class in WTL, `CCommandBarCtrl`[1], has a method called `TrackPopupMenu` that you can use to display a context menu, which is nearly identical to the `TrackPopupMenuEx` Win32 API function:

```
BOOL
CCommandBarCtrlImpl::TrackPopupMenu(
   HMENU hMenu,
   UINT uFlags,
   int x,
   int y,
   LPTPMPARAMS lpParams = NULL)
```

[1] CCommandBarCtrl is defined in atlctlw.h.

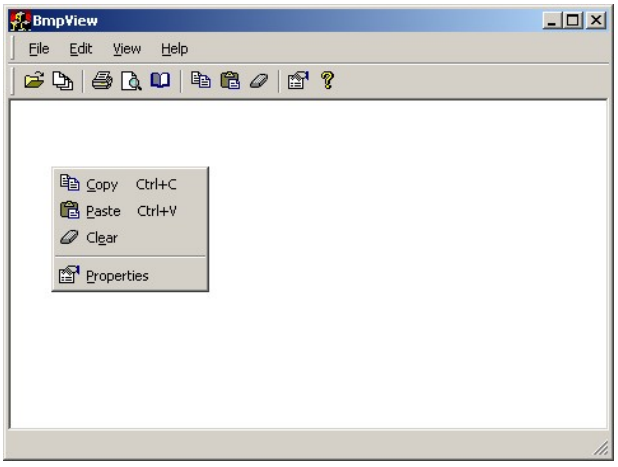
Before I can use the `TrackPopupMenu` method, I need to load the menu resource and get the popup menu portion of it to display our context menu. While all of that can be done using Win32 API and menu handles, it is more convenient to have a class that wraps a menu. Luckily, WTL provides such a class. Just like the GDI classes discussed in Part I, there are two classes that wrap an `HMENU`: `CMenu` and `CMenuHandle`. `CMenu` is the managed version and destroys the `HMENU` when it goes out of scope. `CMenuHandle` is unmanaged and does nothing in its destructor.

Let's now see how I use all of this for our context menu. In the main window class, `CMainFrame`, I'll handle the `WM_CONTEXTMENU` message. If the message comes from the view window, I'll load and display the context menu. Here is the code:

```
LRESULT CMainFrame::OnContextMenu(
UINT, WPARAM wParam, LPARAM lParam, BOOL& bHandled) {
  if((HWND)wParam == m_view) {
    CMenu menu;
    menu.LoadMenu(IDR_CONTEXTMENU);
    CMenuHandle menuPopup = menu.GetSubMenu(POPUP_MENU_POSITION);
    m_CmdBar.TrackPopupMenu(menuPopup,
              TPM_RIGHTBUTTON | TPM_VERPOSANIMATION | TPM_VERTICAL,
              GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
  }
  else bHandled = FALSE;

  return 0;
}
```

You can see that I used `CMenu` class to load the menu resource. That menu will be automatically destroyed when the `CMenu` object goes out of scope. To get the popup menu, I used the unmanaged version, `CMenuHandle`. Because the popup menu is a part of the parent menu, there is no need to destroy it separately. When I have the menu loaded, I use the command bar object, `m_CmdBar` to call its `TrackPopupMenu` method to display the context menu. Because the command bar has all the menu item images loaded, it will display them next to the appropriate menu item. The app with the context menu, including the associated images, is shown in **Figure 2**.

Figure 2: Application with Context Menu



Now I have the code that has all of the UI elements that I need. I can proceed to write the code to display bitmaps, which is, after all, the main point of the sample.

Scrolling Views

The WTL App Wizard created a view class for us. I will use that class to display bitmaps. The App Wizard generated class has only an empty `OnPaint` handler, and that's close to what I need. I just need to implement the `OnPaint` handler to display the bitmap. However, bitmaps can be of any size, and that size can be bigger than the window I have. I need to allow users to scroll and see the whole bitmap.

Scrolling allows you to view different portions of the window than display by default. Some of the standard Windows controls already provide support for this, so you don't have to do any additional work. But, if you draw the content of the window yourself like I am, you need to write code to support scrolling. WTL provides classes to help you implement a window that has built in support for scrolling and makes it much easier to draw without a regard to what is the current visible area. The classes let you concentrate only on the presentation area that you want to draw, and ignore the current scroll state of the window.

WTL provides two classes for the basic scrolling support: `CScrollImpl`, and `CScrollWindowImpl`. `CScrollImpl` is a template class that provides generic scrolling support that you can add to any window class by deriving from `CScrollImpl` and chaining to its message map. `CScrollWindowImpl` derives from `CScrollImpl` and `CWindowImpl`, as well as supplying a message map to provide a full scrolling window implementation. If you derive from `CScrollWindowImpl`, you don't also need to derive from some form of `CWindowImpl`.

WTL also provides another two classes: `CMapScrollImpl` and `CMapScrollWindowImpl`, which, in addition to scrolling, also support mapping modes for drawing. These classes have the same usage as the basic ones; `CMapScrollImpl` derives from `CScrollImpl` and adds support for the mapping modes; `CMapScrollWindowImpl` adds a message map for the full-window implementation. Support for mapping modes allows you to easily use the coordinate system and units that are the best fit for your content.

Regardless of your scrolling implementation base class, you still need to do two things: set the size of your content (so the scrolling can happen properly), and do the actual drawing (no amount of template based classes will ever save you from that). However, you do not have to handle the `WM_PAINT` message or worry about which part of the content of your window to draw, relative to the current scroll position. Instead, you set the `SetScrollSize` and implement the `DoPaint` method. Setting the size of the scroll region is done with any number of overloaded `CScrollImpl` helper functions called `SetScrollSize`:

```
void SetScrollSize(int xMin, int yMin, int xMax, int yMax,
                   BOOL bRedraw = TRUE);
void SetScrollSize(RECT& rcScroll, BOOL bRedraw = TRUE);
void SetScrollSize(int cx, int cy, BOOL bRedraw = TRUE);
void SetScrollSize(SIZE size, BOOL bRedraw = NULL[2]);
```

[2] Yep. That last parameter defaults to NULL instead of TRUE. Unfortunately, this will cause bRedraw to default to FALSE instead of TRUE when using a SIZE structure.

You should call `SetScrollSize` whenever the size of your content changes. In the BmpView sample, I do this when the bitmap to display changes:

```
class CBitmapView : public CScrollWindowImpl<CBitmapView> {
…
  void SetBitmap(HBITMAP hBitmap) {
    if(!m_bmp.IsNull()) m_bmp.DeleteObject();

    m_bmp = hBitmap;

    if(!m_bmp.IsNull()) m_bmp.GetSize(m_size);
    else m_size.cx = m_size.cy = 1;

    SetScrollOffset(0, 0, FALSE);
    SetScrollSize(m_size);
  }
…
};
```

When it's time to draw, remember that `CScrollWindowImpl` handles the `WM_PAINT` message and forwards it to our `DoPaint` function. `DoPaint` will be called with the appropriate Win32 viewport settings to allow you to draw your content with the upper left hand corner adjusted for the current scroll position. It's not really so complicated as it seems, as shown in the `CScrollImpl` class's implementation of `OnPaint`:

```
LRESULT CScrollImpl::OnPaint(UINT, WPARAM wParam, LPARAM, BOOL&) {
   T* pT = static_cast<T*>(this);
   ATLASSERT(::IsWindow(pT->m_hWnd));
   if(wParam != NULL) { // The HDC is sometimes passed in
         CDCHandle dc = (HDC)wParam;
         dc.SetViewportOrg(-m_ptOffset.x, -m_ptOffset.y);
         pT->DoPaint(dc);
   }
   else {
         CPaintDC dc(pT->m_hWnd);
         dc.SetViewportOrg(-m_ptOffset.x, -m_ptOffset.y);
         pT->DoPaint(CDCHandle(dc));
   }
   return 0;
}
```

Our view's implementation of `DoPaint` does pretty much what you'd expect:

```
class CBitmapView : public CScrollWindowImpl<CBitmapView> {
…
  void DoPaint(CDCHandle dc) {
    if(!m_bmp.IsNull()) {
      CDC dcMem;
      dcMem.CreateCompatibleDC(dc);
      HBITMAP hBmpOld = dcMem.SelectBitmap(m_bmp);
      dc.BitBlt(0, 0, m_size.cx, m_size.cy, dcMem, 0, 0, SRCCOPY);
      dcMem.SelectBitmap(hBmpOld);
```

```
        }
      }
    …
    };
```

So far, so good—I now have the view window to display our bitmap. Of course, before I can display a bitmap, I first have to be told the bitmap to display, which is perfect for WTL's `CFileDialog` class.

Common Dialogs

Windows provides a set of common dialogs to make developing applications easier, and also to provide consistency between Windows applications. They represent standard UI services for choosing a file name, font, color, etc. Naturally, WTL includes a complete set of classes that encapsulate Windows common dialogs; they are defined in atldlgs.h. The classes make it easier to create common dialogs, set initial data, and retrieve values from them. WTL classes support all of the common dialogs, including the new Windows 2000 specific ones (as shown in **Table 1**).

Table 1: WTL Common Dialog Classes

WTL Wrapper Class Name	Corresponding Win32 Function Name
CFileDialog	GetOpenFileName and GetSaveFileName
CFolderDialog	SHBrowseForFolder
CFontDialog	ChooseFont
CRichEditFontDialog	ChooseFont
CColorDialog	ChooseColor
CPrintDialog	PrintDlg
CPrintDialogEx	PrintDlgEx
CPageSetupDialog	PageSetupDlg
CFindReplaceDialog	FindText and ReplaceText

Use of common dialogs usually involves setting data to an appropriate structure, then passing that structure in a call to a Windows API to display the common dialog. If more customization of the dialog is needed, developers must write a hook procedure that will be called when a common dialog receives messages. All of this becomes much easier with WTL common dialog classes. Structures are automatically filled with default values you can override either through a constructor argument or through direct access. Most of the common dialog classes implement method called `DoModal` to bring up the common dialog. They also implement methods that send messages to common dialogs, or retrieve values that users set in the dialog. All of the common dialog classes allow you to simply derive from them and implement a message map, override a function, or to customize the behavior of the dialog. Thus, WTL common dialog classes make use and customization of common dialogs much more pleasant.

BmpView needs to use the File Open common dialog. The WTL class, `CFileDialog`, wraps the Windows functions `GetOpenFileName` and `GetSaveFileName`. I will display this dialog in response to the `ID_FILE_OPEN` command in our sample:

```
LRESULT CMainFrame::OnFileOpen(WORD, WORD, HWND, BOOL&) {
  // Passing TRUE as the first ctor arg specifies
  // the File Open dialog instead of the File Save dialog.
  CFileDialog dlg(TRUE, _T("bmp"), NULL,
            OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
            _T("Bitmap Files (*.bmp)\0")
            _T("*.bmp\0All Files (*.*)\0*.*\0"),
            m_hWnd);
```

```
    // All modal common dialog wrapper classes have DoModal
    if(dlg.DoModal() == IDOK) {
      HBITMAP hBmp = (HBITMAP)::LoadImage(NULL, dlg.m_szFileName,
                                  IMAGE_BITMAP, 0, 0,
                                  LR_DEFAULTCOLOR | LR_LOADFROMFILE);

      m_view.SetBitmap(hBmp);
    }

    return 0;
  }
```

You can see that the class simplifies the use of the `GetOpenFileName` common dialog. I don't need to set all members of the `OPENFILENAME` structure directly. Instead, the settings are passed as arguments to `CFileDialog` constructor.

For loading the bitmap from the specified file, I'll use `LoadImage` Windows API, which can load a bitmap from a file. If load succeeds, I'll just set its handle to the data member in the view class, and the view class will paint the bitmap. **Figure 3** presents our program with a loaded bitmap.

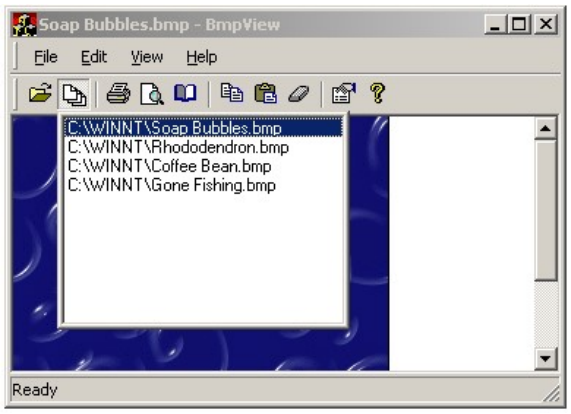
Figure 3: Program with Loaded Bitmap

It would be nice if our program could remember which bitmaps were loaded recently, so I could reload them without navigating around the disk and finding the location of the exact file. For that I will need the most recently used (MRU) list of files. Recall from Part I that I discussed the `CRecentDocumentList`. The BmpView sample's `CMainFrame` class shows the use of the MRU help class if you want to examine another example. For BmpView, it would also be fun to add the same MRU list to a toolbar button. To do that, I will display a popup list box and populate it with items from the MRU list.

Control Wrappers

Figure 4 shows BmpView with the MRU list dropped down from the toolbar.

Figure 4: Bmp view with Toolbar's MRU List



I'm implementing this feature with a listbox control. Normally, working with a listbox or any of the standard or common Windows controls consists of a bunch of SendMessage calls like so:

```
BOOL CMainFrame::BuildList(HWND lb, CRecentDocumentList& mru)
{
  ::SendMessage(lb, LB_RESETCONTENT, 0, 0);

  int nSize = mru.m_arrDocs.GetSize();
  for(int i = 0; i < nSize; i++)
    ::SendMessage(lb, LB_INSERTSTRING, (WPARAM)-1,
                  (LPARAM)mru.m_arrDocs[i].szDocName);

  if(nSize > 0) {
    ::SendMessage(lb, LB_SETCURSEL, 0, 0);
    ::SendMessage(lb, LB_SETTOPINDEX, 0, 0);
  }

  return TRUE;
}
```

At this point, I am really using `SendMessage` as a universal way to call a Windows control's "member functions," but only in a very C-like way. Of course, the standard C++ programmer reflex to wrap such things in a C++ class produces wrapper classes like the following:

```
class CListBox : CWindow {
public:
  void ResetContent() {
    ::SendMessage(m_hWnd, LB_RESETCONTENT, 0, 0L);
  }
  int InsertString(int nIndex, LPCTSTR lpszItem) {
    return (int)::SendMessage(m_hWnd, LB_INSERTSTRING, nIndex,
                              (LPARAM)lpszItem);
  }
  int SetCurSel(int nSelect) {
    return (int)::SendMessage(m_hWnd, LB_SETCURSEL, nSelect, 0L);
  }
  int SetTopIndex(int nIndex) {
    return (int)::SendMessage(m_hWnd, LB_SETTOPINDEX, nIndex, 0L);
  }
…
};
```

This allows me to write the following code instead:

```
BOOL CMainFrame::BuildList(HWND hwndLB, CRecentDocumentList& mru)
{
  CListBox lb(hwndLB);
  lb.ResetContent();

  int nSize = mru.m_arrDocs.GetSize();
  for(int i = 0; i < nSize; i++)
    lb.InsertString(-1, mru.m_arrDocs[i].szDocName);

  if(nSize > 0) {
    lb.SetCurSel(0);
    lb.SetTopIndex(0);
  }

  return TRUE;
}
```

Of course, WTL provides a full compliment of such wrappers, as defined in atlctls.h and listed in **Table 2**.

Table 2: WTL Control Wrapper Classes

WTL Wrapper Class Name	Corresponding Win32 Window Class
CStatic	STATIC
CButton	BUTTON
CListBox	LISTBOX
CComboBox	COMBOBOX
CEdit	EDIT
CScrollBar	SCROLLBAR
CToolTipCtrl	tooltips_class32
CListViewCtrl	SysListView32
CTreeViewCtrl	SysTreeView32
CHeaderCtrl	SysHeader32
CToolBarCtrl	ToolbarWindow32
CStatusBarCtrl	msctls_statusbar32
CTabCtrl	SysTabControl32
CTrackBarCtrl	msctls_trackbar32
CUpDownCtrl	msctls_updown32
CProgressBarCtrl	msctls_progress32
CHotKeyCtrl	msctls_hotkey32
CAnimateCtrl	SysAnimate32
CRichEditCtrl	RichEdit20A or RichEdit20W
CDragListBox	LISTBOX
CReBarCtrl	ReBarWindow32
CComboBoxEx	ComboBoxEx32
CDateTimePickerCtrl	SysDateTimePick32
CMonthCalendarCtrl	SysMonthCal32
CIPAddressCtrl	SysIPAddress32
CPagerCtrl	SysPager

The control wrapper classes are implemented in the same way that the `CWindow` from ATL is; for example, by providing a thin wrapper that gives you the methods to call instead of sending messages. These classes all contain only one data member, a handle to the control's window and short inline functions. The classes simplify usage of controls, while not adding any baggage. With the compiler optimizations turned on in the release builds, the resulting code is usually the same as using handles and messages directly.

To implement our toolbar MRU, I will use the `CListBox` class. I could just create an instance of `CListBox`, but how would I handle the selection? I'd really like a way to be able to handle the listbox's messages in a self-contained way and send a `WM_COMMAND` messages when an item from the MRU is selected. By being a little bit fancy, I can use the 2nd parameter to the `CWindowImpl` template to solve my problem. Recall ATL's `CWindowImpl` class declaration:

```
template <class T,
          class TBase = CWindow,
          class TWinTraits = CControlWinTraits>
class ATL_NO_VTABLE CWindowImpl :
    public CWindowImplBaseT< TBase, TWinTraits > {…};
```

The `TBase` parameter to the `CWindowImpl` class allows me to change the base class from a do-nothing window to anything I like; e.g., the `CListBox` class in my case. This C++ inheritance trick is also effectively doing Win32 superclassing, allowing me to handle the LISTBOX class's window messages before or instead of letting the LISTBOX do it. That's all I need to implement our special-purpose MRU listbox class:

```
class CMruList : public CWindowImpl<CMruList, CListBox> {
public:
  SIZE m_size;

  CMruList() {
    m_size.cx = 200;
    m_size.cy = 150;
  }

  HWND Create(HWND hWndParent) {
    CWindowImpl<CMruList, CListBox>::Create(hWndParent, rcDefault,
      NULL,
      WS_POPUP | WS_THICKFRAME | WS_CLIPCHILDREN | WS_CLIPSIBLINGS |
      WS_VSCROLL | LBS_NOINTEGRALHEIGHT,
      WS_EX_CLIENTEDGE);

    if(IsWindow()) SetFont(AtlGetStockFont(DEFAULT_GUI_FONT));

    return m_hWnd;
  }

  BOOL BuildList(CRecentDocumentList& mru) {
    ATLASSERT(IsWindow());

    ResetContent();

    int nSize = mru.m_arrDocs.GetSize();
    for(int i = 0; i < nSize; i++)
      // docs are in reversed order in the array
      InsertString(0, mru.m_arrDocs[i].szDocName);

    if(nSize > 0) {
      SetCurSel(0);
      SetTopIndex(0);
    }

    return TRUE;
  }

  BOOL ShowList(int x, int y) {
    return SetWindowPos(NULL, x, y, m_size.cx, m_size.cy,
                        SWP_NOZORDER | SWP_SHOWWINDOW);
  }

  void HideList() {
    RECT rect;
```

```
  GetWindowRect(&rect);
  m_size.cx = rect.right - rect.left;
  m_size.cy = rect.bottom - rect.top;
  ShowWindow(SW_HIDE);
}

void FireCommand() {
  int nSel = GetCurSel();
  if(nSel != LB_ERR) {
    ::SetFocus(GetParent());  // will hide this window
    ::SendMessage(GetParent(), WM_COMMAND,
         MAKEWPARAM((WORD)(ID_FILE_MRU_FIRST + nSel), LBN_DBLCLK),
         (LPARAM)m_hWnd);
  }
}

BEGIN_MSG_MAP(CMruList)
  MESSAGE_HANDLER(WM_KEYDOWN, OnKeyDown)
  MESSAGE_HANDLER(WM_LBUTTONDBLCLK, OnLButtonDblClk)
  MESSAGE_HANDLER(WM_KILLFOCUS, OnKillFocus)
  MESSAGE_HANDLER(WM_NCHITTEST, OnNcHitTest)
END_MSG_MAP()

LRESULT OnKeyDown(UINT, WPARAM wParam, LPARAM, BOOL& bHandled) {
  if(wParam == VK_RETURN) FireCommand();
  else bHandled = FALSE;
  return 0;
}

LRESULT OnLButtonDblClk(UINT, WPARAM, LPARAM, BOOL&) {
  FireCommand();
  return 0;
}

LRESULT OnKillFocus(UINT, WPARAM, LPARAM, BOOL&) {
  HideList();
  return 0;
}

LRESULT
OnNcHitTest(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&) {
  LRESULT lRet = DefWindowProc(uMsg, wParam, lParam);
  switch(lRet)
  {
  case HTLEFT:
  case HTTOP:
  case HTTOPLEFT:
  case HTTOPRIGHT:
  case HTBOTTOMLEFT:
    lRet = HTCLIENT;  // don't allow resizing here
    break;
```

```
      default:
        break;
      }
      return lRet;
    }
};
```

Bending a listbox to our will is fun, but not too hard. Something that is hard under Windows is printing and print preview, which I cover next.

Printing and Print Preview

MFC applications that support the document-view model get printing and print preview nearly for free as part of the `CView` class. However, if you use MFC but not document-view, you're pretty much on your own. WTL's support, while not so seamless, does support printing and print preview regardless of whether you have a view class at all.

The main WTL class that coordinates printing in an application, as defined in atlprint.h, is `CPrintJob`. It provides a method called `StartPrintJob` that you call to start your printing. In that call you specify if your printing should be in the background, which printer you will use, the default printer settings, etc. You also have to pass an object that supports the `IPrintJobInfo` interface, in which you have to implement the actual code that does printing of each page.

```
// Defines callbacks used by CPrintJob
// This is not a COM interface
class ATL_NO_VTABLE IPrintJobInfo {
public:
  virtual void BeginPrintJob(HDC hDC)=0;
  virtual void EndPrintJob(HDC hDC, bool bAborted)=0;
  virtual void PrePrintPage(UINT nPage, HDC hDC)=0;
  virtual bool PrintPage(UINT nPage, HDC hDC)=0;
  virtual void PostPrintPage(UINT nPage, HDC hDC)=0;
  virtual DEVMODE* GetNewDevModeForPage(UINT nLastPage,
                                        UINT nPage)=0;
  virtual bool IsValidPage(UINT nPage)[3];
};
```

`IPrintJobInfo` is a C++ interface and specifies methods that will be called by `CPrintJob`. WTL also includes the `CPrintJobInfo` class that derives from `IPrintJobInfo` and provides a default implementation for it. It implements every method of `IPrintJobInfo` but `PrintPage`. If you don't need extra customization per page, you can just derive your class from `CPrintJobInfo` and implement `PrintPage`. If you override other methods of the `CPrintJobInfo` class, you can completely customize printing on the page basis, for instance, provide different printer settings for different pages. WTL also provides several more classes to help with printing: `CDevModeT`, `CPrinterT`, `CPrinterDC`, and `CPrinterInfo`. These classes are implemented as wrapper classes to help with using Windows printing API.

The simplest way to implement printing in your application is to create a `CPrintJob` object, call `StartPrintJob`, and pass an object of a class derived from `CPrintJobInfo`. The `CPrintJob` object will coordinate printing and will call methods of the `CPrintJobInfo` derived object to do the actual printing.

[3] IsValidPage was added last, which is probably why Microsoft forgot to add the =0 at the end.

I will derive my main window class, `CMainFrame`, from `CPrintJobInfo` and implement required callbacks there. First, I need a few new data members in the `CMainFrame` class:

```
class CMainFrame : …, public CPrintJobInfo {
…
  CPrinter m_printer;
  CDevMode m_devmode;
  RECT     m_rcMargin;
};
```

Next, I will implement these two callback required for printing:

```
bool CMainFrame::IsValidPage(UINT nPage) {
  return (nPage == 0);  // I have only one page
}

bool CMainFrame::PrintPage(UINT nPage, HDC hDC) {
  if (nPage >= 1) return false; // I have only one page

  ATLASSERT(!m_view.m_bmp.IsNull());

  …

  // Print
  dc.StretchBlt(xOff, yOff, cxBlt, cyBlt, dcMem, 0, 0, cx, cy,
                SRCCOPY);

  return true;
}
```

`IsValidPage` tells the printing code if the page is still in the range that I want to print. Its implementation here is very simple, since I have only one page that to print. `PrintPage` does the actual work, and it stretches the bitmap to fit most of the page. Other schemes are certainly possible, of course.

Just to be nice, I'll allow users to specify the printer that they want to print on, as well as to allow them to specify the page layout. This just means I need two more common dialogs: print and page layout. Support for them in WTL comes in two classes: `CPrintDialog` and `CPageSetup` dialog. Just like `CFileDialog` class, they simplify the usage of the common dialogs by allowing you to pass arguments to the constructor of the class, then use `DoModal` method to display the dialog itself. I will use them in `ID_FILE_PRINT` and `ID_FILEPAGE_SETUP` command handlers, shown here:

```
LRESULT CMainFrame::OnFilePrint(WORD, WORD, HWND, BOOL&) {
  CPrintDialog dlg(FALSE);
  dlg.m_pd.hDevMode = m_devmode.CopyToHDEVMODE();
  dlg.m_pd.hDevNames = m_printer.CopyToHDEVNAMES();
  dlg.m_pd.nMinPage = 1;
  dlg.m_pd.nMaxPage = 1;

  if(dlg.DoModal() == IDOK) {
    m_devmode.CopyFromHDEVMODE(dlg.m_pd.hDevMode);
    m_printer.ClosePrinter();
```

```
    m_printer.OpenPrinter(dlg.m_pd.hDevNames, m_devmode.m_pDevMode);

    CPrintJob job;
    job.StartPrintJob(false, m_printer, m_devmode.m_pDevMode, this,
                      _T("BmpView Document"), 0, 0);
  }

  ::GlobalFree(dlg.m_pd.hDevMode);
  ::GlobalFree(dlg.m_pd.hDevNames);

  return 0;
}

LRESULT CMainFrame::OnFilePageSetup(WORD, WORD, HWND, BOOL&) {
  CPageSetupDialog dlg;
  dlg.m_psd.hDevMode = m_devmode.CopyToHDEVMODE();
  dlg.m_psd.hDevNames = m_printer.CopyToHDEVNAMES();
  dlg.m_psd.rtMargin = m_rcMargin;

  if(dlg.DoModal() == IDOK)
  {
    if(m_bPrintPreview) TogglePrintPreview(); // Later…

    m_devmode.CopyFromHDEVMODE(dlg.m_psd.hDevMode);
    m_printer.ClosePrinter();
    m_printer.OpenPrinter(dlg.m_psd.hDevNames,
                          m_devmode.m_pDevMode);
    m_rcMargin = dlg.m_psd.rtMargin;
  }

  ::GlobalFree(dlg.m_psd.hDevMode);
  ::GlobalFree(dlg.m_psd.hDevNames);

  return 0;
}
```

Support for print preview brings three more classes: `CPrintPreview`, `CPrintPreviewWindowImpl`, and `CPrintPreviewWindow`. `CPrintPreview` is a base class that provides the basic print preview implementation. It contains the `SetPrintPreviewInfo` method, which is similar to `StartPrintJob` of the `CPrintJob` class. It also accepts arguments for the target printer, default DEVMODE, and an `IPrintJobInfo` callback interface. `CPrintPreview` creates an enhanced metafile and you can call the `DoPaint` method to display it in the window.

You can derive your class from `CPrintPreview` and from a window implementation class to create a class for the print preview window. Or, you can use already provided classes in WTL: `CPrintPreviewWindowImpl` and `CPrintPreviewWindow`. `CPrintPreviewWindowImpl` is a template class that implements a print preview window. You need to derive from it and, optionally, customize its behavior or add more features by handling messages. If you don't need any customization, you can use `CPrintPreviewWindow`, which is a ready-to-use class that implements a complete print preview window.

I will switch to the print preview mode and back by handling the `ID_FILE_PRINT_PREVIEW` command and calling the `TogglePrintPreview` helper function:

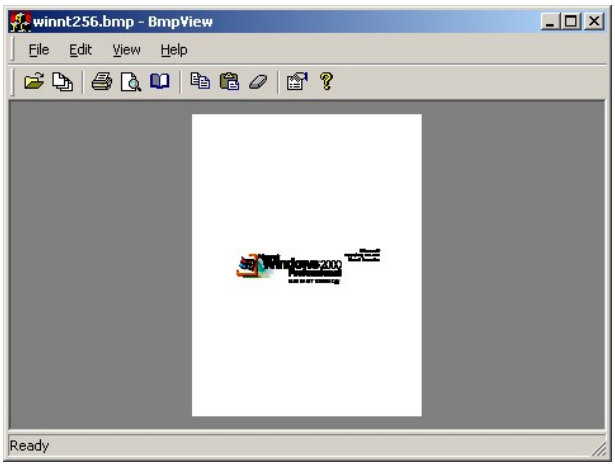
```
LRESULT CMainFrame::OnFilePrintPreview(WORD, WORD, HWND, BOOL&) {
  TogglePrintPreview();
  return 0;
}
```

`TogglePrintPreview` implementation looks like this:

```
void CMainFrame::TogglePrintPreview() {
  if(m_bPrintPreview) { // close the print preview window
    ATLASSERT(m_hWndClient == m_wndPreview.m_hWnd);

    m_hWndClient = m_view;
    m_view.ShowWindow(SW_SHOW);
    m_wndPreview.DestroyWindow();
  }
  else { // display the print preview window
    ATLASSERT(m_hWndClient == m_view.m_hWnd);

    m_wndPreview.SetPrintPreviewInfo(m_printer,
                                     m_devmode.m_pDevMode,
                                     this, 0, 0);
    m_wndPreview.SetPage(0);

    m_wndPreview.Create(m_hWnd, rcDefault, NULL, 0,
                        WS_EX_CLIENTEDGE);
    m_view.ShowWindow(SW_HIDE);
    m_hWndClient = m_wndPreview;
  }

  m_bPrintPreview = !m_bPrintPreview;
  UpdateLayout();
}
```

That's it! Now BmpView has the full support for printing and print preview. The application's print preview is shown in the **Figure 5**.

Figure 5: Application's Print Preview



There are also other ways to use WTL's printing support. More advanced features of `CPrintJob` allow you to do background printing on a different thread, and cancel the print job while in progress. For background printing, the only difference is that the call to `StartPrintJob` returns immediately, and all of the calls to methods of the `IPrintJobInfo` occur on a different thread. You need to ensure that doesn't cause a problem; for instance, that data doesn't change during printing, or that the application doesn't close, etc.

I can now print the bitmaps, and I would also like to display their properties. I could have used a custom dialog for that, but opted for a property sheet since there are several categories of properties that I want to show. (I want to show you how to use Property Sheets in WTL—I'll leave it to you to decide which was the real deciding factor…)

Properties Sheets

Properties that I want to display about our bitmap can be the properties of the bitmap or of the file. I'll toss in the properties of the display as well, then use a property sheet to display those categories on different pages.

Property sheets and pages are special dialogs provided by Windows to help implement property dialogs (also called tabbed dialogs) as well as wizards. They are very commonly used in applications today, so WTL provides classes to encapsulate them and make their implementation and usage easier.

The classes are implemented in three variants: a `HWND` wrapper class (`CPropertySheetWindow` and `CPropertyPageWindow`), a template window implementation class (`CPropertySheetImpl` and `CPropertyPageImpl`), and a ready-to-use windows class (`CPropertySheet` and `CPropertyPage`). These three variants give you flexibility to use or implement a property sheet or page in the most efficient way. I will use template window implementation variants. For each page, I'll derive class for `CPropertyPageImpl` and add page specific code. For the sheet itself, I'll derive a class for `CPropertySheetImpl`. Let's have a look:

```
class CPageOne : public CPropertyPageImpl<CPageOne> {
public:
  enum { IDD = IDD_PROP_PAGE1 }; // Required, just like CDialogImpl

  LPCTSTR m_lpstrFilePath;
  CFileName m_filename;


  CPageOne() : m_lpstrFilePath(NULL)
  { }

  BEGIN_MSG_MAP(CPageOne)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    CHAIN_MSG_MAP(CPropertyPageImpl<CPageOne>)
  END_MSG_MAP()

  LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&);
};

class CPageTwo : public CPropertyPageImpl<CPageTwo> {
public:
  enum { IDD = IDD_PROP_PAGE2 };

  LPCTSTR m_lpstrFilePath;
  CBitmapHandle m_bmp;


  CPageTwo() : m_lpstrFilePath(NULL)
  { }

  BEGIN_MSG_MAP(CPageTwo)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    CHAIN_MSG_MAP(CPropertyPageImpl<CPageTwo>)
  END_MSG_MAP()

  LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&);
};

class CPageThree : public CPropertyPageImpl<CPageThree>
{
public:
  enum { IDD = IDD_PROP_PAGE3 };

  BEGIN_MSG_MAP(CPageThree)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    CHAIN_MSG_MAP(CPropertyPageImpl<CPageThree>)
  END_MSG_MAP()

  LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL&);
};
```

```
class CBmpProperties : public CPropertySheetImpl<CBmpProperties> {
public:
  CPageOne m_page1;
  CPageTwo m_page2;
  CPageThree m_page3;


  CBmpProperties() {
    m_psh.dwFlags |= PSH_NOAPPLYNOW;

    AddPage(m_page1);
    AddPage(m_page2);
    AddPage(m_page3);

    SetActivePage(1);
    SetTitle(_T("Bitmap Properties"));
  }

  void SetFileInfo(LPCTSTR lpstrFilePath, HBITMAP hBitmap);

  BEGIN_MSG_MAP(CBmpProperties)
    CHAIN_MSG_MAP(CPropertySheetImpl<CBmpProperties>)
  END_MSG_MAP()
};
```

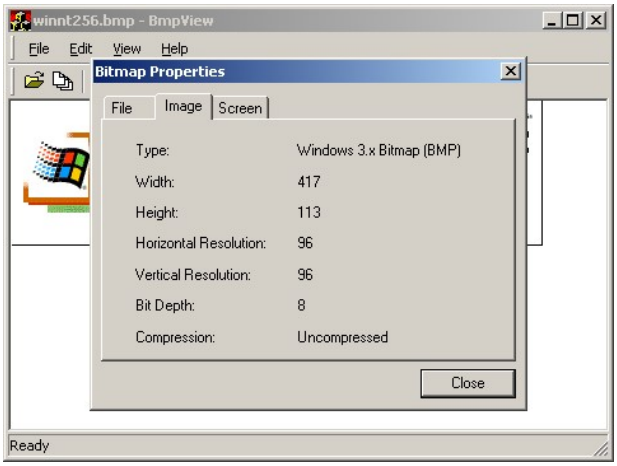
At this point in your ATL/WTL career, there should really be no surprise about how WTL property page and property sheet classes are constructed. The pages derive from the implementation base class, providing a resource ID for the dialog box resource to use (just like a class derived from `CDialogImpl`). To get the default windows messages handled properly, chain to the base class in the message map. The interesting part is the property sheet, which contains one instance each of the three property page classes. On construction, I can manipulate the Win32 property sheet flags (turning off the Apply button in this case) and, most importantly, add each of the C++ page objects to the list of pages to appear when the property sheet is shown. For completeness, I set the first page (I could've pick any of them) and set the title of the property sheet. Likewise, for the default message handling to be handled properly for us, I chain to the base class.

Now I can display properties in response to the `ID_VIEW_PROPERTIES` command by simply creating an instance of our new `CBmpProperties` class and calling the `DoModal` method:

```
LRESULT CMainFrame::OnViewProperties(WORD, WORD, HWND, BOOL&) {
  CBmpProperties prop;

  if(lstrlen(m_szFilePath) > 0)  // I have a file name
    prop.SetFileInfo(m_szFilePath, NULL);
  else        // must be clipboard then
    prop.SetFileInfo(NULL, m_view.m_bmp.m_hBitmap);

  prop.DoModal();
  return 0;
}
```

Figure 6 shows the application with the property sheet displayed.

Figure 6: Application with Property Page



I am almost done with the sample. One important thing is missing, because I can now display bitmaps and have all nice UI elements, I also must have the proper updating of those UI elements do indicate the state that they are in.

Message Filtering

The code generated by the WTL App Wizard uses a module and message loop classes from WTL. Those classes provide a core support for message filtering and idle processing. Message filtering is a handy way to route messages between windows in your application after `GetMessage` pulls off the queue, but before they get processed via `TranslateMessage` and `DispatchMessage`. This is an especially handy place to call, for example, `IsDialogMessage` for a modeless dialog box.

WTL supports message filtering via an extensible class that encapsulates the traditional message pump triad, i.e., `GetMessage/TranslateMessage/DispatchMessage`. The class is called `CMessageLoop` and the wizard generates an instance of one for our use:

```
int Run(LPTSTR = NULL, int nCmdShow = SW_SHOWDEFAULT) {
   CMessageLoop theLoop;
   _Module.AddMessageLoop(&theLoop);

   CMainFrame wndMain;
   if(wndMain.CreateEx() == NULL) return 0;
   wndMain.ShowWindow(nCmdShow);

   int nRet = theLoop.Run();
   _Module.RemoveMessageLoop();

   return nRet;
}

int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE,
                     LPTSTR lpstrCmdLine, int nCmdShow)
{
   INITCOMMONCONTROLSEX iccx;
```

```
   iccx.dwSize = sizeof(iccx);
   iccx.dwICC = ICC_COOL_CLASSES | ICC_BAR_CLASSES;
   BOOL bRet = ::InitCommonControlsEx(&iccx);
   bRet;
   ATLASSERT(bRet);

   HRESULT hRes = _Module.Init(NULL, hInstance);
   hRes;
   ATLASSERT(SUCCEEDED(hRes));

   int nRet = Run(lpstrCmdLine, nCmdShow);

   _Module.Term();
   return nRet;
}
```

Notice that the _Module object has a new call you may not be familiar with: `AddMessageLoop`. This is not your father's _Module. Rather, it's the `CAppModule` that comes with WTL and is defined in atlapp.h:

```
class CAppModule : public CComModule {
public:
   DWORD                                m_dwMainThreadID;
   CSimpleMap<DWORD, CMessageLoop*>*    m_pMsgLoopMap;
   CSimpleArray<HWND>*                  m_pSettingChangeNotify;

// Overrides of CComModule::Init and Term
   HRESULT Init(_ATL_OBJMAP_ENTRY* pObjMap, HINSTANCE hInstance,
                const GUID* pLibID = NULL);
   void Term();

// Message loop map methods
   BOOL AddMessageLoop(CMessageLoop* pMsgLoop);
   BOOL RemoveMessageLoop();
   CMessageLoop* GetMessageLoop(DWORD dwThreadID) const;

// Setting change notify methods
   BOOL AddSettingChangeNotify(HWND hWnd);
   BOOL RemoveSettingChangeNotify(HWND hWnd);
...
};
```

The application module is able to manage one message loop/thread (which is handy when you're implementing multi-SDI). When I call `Run`, the message loop object does pretty much what you'd expect:

```
int CMessageLoop::Run() {
  BOOL bDoIdle = TRUE;
  int  nIdleCount = 0;
  BOOL bRet;

  for(;;) {
    while(!::PeekMessage(&m_msg, NULL, 0, 0, PM_NOREMOVE) &&
```

```
        bDoIdle) {
      if(!OnIdle(nIdleCount++)) bDoIdle = FALSE;
    }

    bRet = ::GetMessage(&m_msg, NULL, 0, 0);

    if(bRet == -1) continue; // error, don't process
    else if(!bRet) break;    // WM_QUIT, exit message loop

    if(!PreTranslateMessage(&m_msg)) {
      ::TranslateMessage(&m_msg);
      ::DispatchMessage(&m_msg);
    }

    if(IsIdleMessage(&m_msg)) {
      bDoIdle = TRUE;
      nIdleCount = 0;
    }
  }

  return (int)m_msg.wParam;
}
```

You can see the call to `GetMessage`, `TranslateMessage`, and `DispatchMessage` just exactly where you expect to see them. There are really just two interesting wrinkles. The second is the idle processing, which I'll cover in a minute. The first, however, is the call to `PreTranslateMessage`. This call walks the list of `CMessageFilter` implementations that have registered themselves for this thread, calling `PreTranslateMessage` until one of them returns TRUE (they've processed the message) or all of them return FALSE (go ahead and translate/dispatch the message). `CMessageFilter` is just a C++ interface:

```
class CMessageFilter {
public:
   virtual BOOL PreTranslateMessage(MSG* pMsg) = 0;
};
```

Notice that the wizard-generated `CMainFrame` class derives from `CMessageFilter` and implements the `PreTranslateMessage` function:

```
class CMainFrame : …, public CMessageFilter {
…
  virtual BOOL PreTranslateMessage(MSG* pMsg) {
    if(CFrameWindowImpl<CMainFrame>::PreTranslateMessage(pMsg))
      return TRUE;
    return m_view.PreTranslateMessage(pMsg);
  }
};
```

The `CMainFrame` first asks the base class if it would like to handle the message, then the view. If I wanted to call `IsDialogMessage` on a modeless dialog, I could do it here, or even better the dialog class itself could derive from `CMessageFilter` and register itself with the _Module using the `AddMessageFilter` member function:

```
class CMyModelessDlg :
  public CDialogImpl<CMyModelessDlg>
  public CMessageFilter {
public:
  CMyModelessDlg() {
    _Module.GetMessageLoop()->AddMessageFilter(this);
  }

  ~CMyModelessDlg() {
    _Module.GetMessageLoop()->RemoveMessageFilter(this);
  }

  virtual BOOL PreTranslateMessage(MSG* pmsg) {
    return IsDialogMessage(m_hWnd, pmsg);
  }
…
};
```

The beauty of the way that message filters work in WTL is that any object can be a message filter and they can be added and removed on the fly, all without tightly coupling one object, like a modeless dialog, to another, like its owner.

Idle Time Processing

The WTL message loop architecture is extensible in another way besides message filtering. It also provides support for idle-time processing. An application is idle when its message queue is empty. When that happens, the `Run` method of `CMessageLoop` notices and calls all of the idle handlers for that thread. An idle handler is merely an implementation of `CIdleHandler` that registers itself with the thread's message loop for callbacks through message filtering:

```
class CIdleHandler {
public:
   virtual BOOL OnIdle() = 0;
};

BOOL CMessageLoop::AddIdleHandler(CIdleHandler* pIdleHandler) {
  return m_aIdleHandler.Add(pIdleHandler);
}

BOOL CMessageLoop::RemoveIdleHandler(CIdleHandler* pIdleHandler) {
  return m_aIdleHandler.Remove(pIdleHandler);
}
```

Any object can register itself as an idle handler and can do whatever it likes during that time. For the BmpView sample, I will use it to add UI updating.

UI Updating

The idea of UI updating (command UI updating) originated in MFC as a way to save the application from constantly calling `EnableMenuItem(TRUE)` when an operation was available and `EnableMenuItem(FALSE)` when it no longer was, regardless of whether the menu with that item has even been shown. The application implements menu UI updating as a callback notification just before it shows a menu item[4]. At that point, the owner of the menu is asked to enable or disable every item in the menu (the default being enable unless otherwise stated). This allows the application programmer to centralize all logic about a particular menu item in one place instead of in every place where the state changed that governed the availability of an operation. The idea grew from there to encompass toolbars, context menus, dialogs, statusbars, etc., all from this simple architecture. WTL could hardly not support it.

WTL's presentation of UI updating is via…, you guessed it, a base class and some map entries[5]. The base class is `CUpdateUI`, which is pretty boring, because all it does is walk a map with entries like I'm using in the `CMainFrame` class:

```
class CMainFrame : …, public CUpdateUI<CMainFrame> {
…
BEGIN_UPDATE_UI_MAP(CMainFrame)
  UPDATE_ELEMENT(ID_FILE_PRINT, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_FILE_PRINT_PREVIEW, \
                 UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_EDIT_COPY, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_EDIT_PASTE, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_EDIT_CLEAR, UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_VIEW_TOOLBAR, UPDUI_MENUPOPUP)
  UPDATE_ELEMENT(ID_VIEW_STATUS_BAR, UPDUI_MENUPOPUP)
  UPDATE_ELEMENT(ID_VIEW_PROPERTIES, \
                 UPDUI_MENUPOPUP | UPDUI_TOOLBAR)
  UPDATE_ELEMENT(ID_RECENT_BTN, UPDUI_TOOLBAR)
END_UPDATE_UI_MAP()
};
```

The update UI map specifies which UI elements will be updated. You can see that I specify elements by their command ID and that I specify what type of UI element they are (menu item, toolbar button, etc.). Some commands can be part of multiple controls. For example `ID_EDIT_COPY` is present on the menu and on the toolbar, so I use two flags, `UPDUI_MENUPOPUP` and `UPDUI_TOOLBAR`, whereas `ID_VIEW_TOOLBAR` only shows up on the menu, so I use just `UPDUI_MENUPOPUP`. WTL supports the following UI element types for UI updating: `UPDUI_MENUPOPUP`, `UPDUI_MENUBAR`, `UPDUI_CHILDWINDOW`, `UPDUI_TOOLBAR`, and `UPDUI_STATUSBAR`.

[4] This is implemented via the WM_INITMENUPOPUP notification message.

[5] The ATL folks are nothing if not consistent.

WTL's UI updating is a bit more manual than MFC's. Instead of `WM_INITMENUPOPUP` messages and/or idle processing automatically kicking in, with WTL you're in control. It turns out that WTL's idle processing allows me to fit in my UI updating very easily by simply registering our `CMainFrame` as an idle handler and implementing `OnIdle` to update the UI like so:

```
BOOL CMainFrame::OnIdle() {
  BOOL bEnable = !m_view.m_bmp.IsNull();

  UIEnable(ID_FILE_PRINT, bEnable);
  UIEnable(ID_FILE_PRINT_PREVIEW, bEnable);
  UISetCheck(ID_FILE_PRINT_PREVIEW, m_bPrintPreview);
  UIEnable(ID_EDIT_COPY, bEnable);
  UIEnable(ID_EDIT_PASTE, ::IsClipboardFormatAvailable(CF_BITMAP));
  UIEnable(ID_EDIT_CLEAR, bEnable);
  UIEnable(ID_VIEW_PROPERTIES, bEnable);
  UISetCheck(ID_RECENT_BTN, m_list.IsWindowVisible());
  UIUpdateToolBar();

  return FALSE;
}
```

In `OnIdle` I check the appropriate state (whether I have a bitmap to display or not) and change the state of the corresponding UI elements appropriately. WTL provides an implementation that doesn't access the real Windows control unless the requested state changes, so executing this code in `OnIdle` will not be expensive. WTL's UI updating class supports the following methods:

```
BOOL UIEnable(int nID, BOOL bEnable, BOOL bForceUpdate = FALSE);
BOOL UISetCheck(int nID, int nCheck, BOOL bForceUpdate = FALSE);
BOOL UISetRadio(int nID, BOOL bRadio, BOOL bForceUpdate = FALSE);
BOOL UISetText(int nID, LPCTSTR lpstr, BOOL bForceUpdate = FALSE);
BOOL UIUpdateMenuBar(BOOL bForceUpdate = FALSE, BOOL bMain = FALSE);
BOOL UIUpdateToolBar(BOOL bForceUpdate = FALSE);
BOOL UIUpdateStatusBar(BOOL bForceUpdate = FALSE);
BOOL UIUpdateChildWindows(BOOL bForceUpdate = FALSE);
BOOL UISetState(int nID, DWORD dwState);
DWORD UIGetState(int nID);
```

The states that `UISetState` and `UIGetState` support are `UPDUI_ENABLED`, `UPDUI_DISABLED`, `UPDUI_CHECKED`, `UPDUI_CHECKED2`, `UPDUI_RADIO`, `UPDUI_DEFAULT`, and `UPDUI_TEXT`. The CUpdateUI class and supporting constants are defined in atlframe.h. Our sample, BmpView is now complete. It does all the tasks that I want it to do, and presents a complete user interface for executing them. The full source code for this sample is included.

Message Cracking

WTL has another feature that, while I didn't need it for our sample, you may find interesting. You may have noticed that in my sample, my message handlers had to "crack" my own messages. Message cracking is the process of taking the WPARAM and LPARAM parameters of a message and, based on the message ID, unpacking type-safe data for more convenient processing. This is really the reverse of the process I have to go through to send messages and for which WTL provides inline message functions in the control wrapper classes. For example, in my CMainFrame's OnContextMenu method, recall that I cracked the x and y parameters out of the LPARAM via a couple of macros:

```
m_CmdBar.TrackPopupMenu(…, GET_X_LPARAM(lParam),
                           GET_Y_LPARAM(lParam));
```

These macros are defined in windowx.h, an ancient header file for C programmers, which can also be used in ATL applications. As ex-MFC programmers, I am surprised that I don't mind cracking messages, but it turns out not to be a problem. However, if you'd prefer to have your messages cracked for you, WTL does provide support for it.

WTL provides an alternative set of macros for ATL message maps. These macros are message specific, and they extract the message specific data. They require handler functions with specific arguments, but unlike MFC, they allow you to specify the name of the handler function.

These macros live in the atlcrack.h file. In there you'll find a macro for each Windows message, and also macros related to notifications sent by controls. You should simply use the macro associated with a particular message that you want to handle and put it in your message map instead of `MESSAGE_HANDLER`. You also implement the handler function according to the required signature for that message. The only difference is that your message map must use the `BEGIN_MSG_MAP_EX` macro instead of the usual `BEGIN_MSG_MAP`. `BEGIN_MSG_MAP_EX` provides a way for "cracked" handlers to retrieve the current message and specify if the message was handled or not.

You will notice that "cracked" handlers do not have a Boolean `bHandled` argument that is present in all raw handlers. So, how do you specify that the message was not handled? `BEGIN_MSG_MAP_EX` defines an additional method, `SetMessageHandled`, which you can use for that purpose. By default, all messages are considered handed if there is a message handler for them. If you want to handle a message, but want to indicate that it wasn't handled, you just need to call `SetMessageHandled` and pass `FALSE` to it.

Here is an example of a simple class that handles WM_CREATE and WM_SIZE messages using ATL's raw handlers:

```
class CMyWindow : public CWindowImpl<CMyWindow> {
public:
  BEGIN_MSG_MAP(CMyWindow)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
    MESSAGE_HANDLER(WM_SIZE, OnSize)
  END_MSG_MAP()

  LRESULT OnCreate(UINT, WPARAM, LPARAM lParam, BOOL&) {
    LPCREATESTRUCT lpCreateStruct = LPCREATESTRUCT(lParam);
    …
    return 0;
  }

  LRESULT OnSize(UINT, WPARAM wParam, LPARAM lParam, BOOL&) {
    UINT nType =  (UINT)wParam;
    CSize size(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
    if(nType != SIZE_MINIMIZED) …
    …
    return 0;
  }
};
```

Now, let's see how the same class would look like if I use cracked message handlers:

```
class CMyWindow : public CWindowImpl<CMyWindow> {
public:
  BEGIN_MSG_MAP_EX(CMyWindow)
    MSG_WM_CREATE(OnCreate)
    MSG_WM_SIZE(OnSize)
  END_MSG_MAP()

  int OnCreate(LPCREATESTRUCT lpCreateStruct) {
    …
    return 0;
  }

  void OnSize(UINT nType, CSize size) {
    if(nType != SIZE_MINIMIZED) …
  }
};
```

You see that in this second version, handlers have arguments that are already expanded so no additional casting is needed. Also, return values exist only is messages that use them.

One downside about using these macros is that WTL doesn't provide a wizard that can generate required function signatures for each message. A wizard like that would make the usage of these macros much easier. This way, you will have to take a look in the macro declaration for the function signature of the handler function.

Dharma Shukla, Chris Sells,
and Nenad Stefanovic

white paper
DEVELOPMENTOR™
www.develop.com

The Future of WTL

Because WTL is an SDK sample and officially undocumented and unsupported by Microsoft, you may wonder at the wisdom of using it in your applications. Put your fears to rest. WTL is merely an extension to ATL in such a logical direction that it's hard to remember where ATL leaves off and WTL begins. Because of that and because of the love that the ATL community has for the ATL/WTL style of programming, WTL is going to live a long and healthy life. Microsoft has been using early versions of WTL internally for years because it produces such small, efficient applications.

Virtual Machines may come and Assembly Language may go, but there will always be a need for the smallest, faster thing you can build. So, while Microsoft doesn't document or support WTL, we will—the "we" being the ATL community at large. Three members of the ATL community have documented WTL in this two-part article. At least one member has provided documentation over at VCDJ. The ATL mailing list (available for subscription at http://discuss.microsoft.com/atl) has been alive with WTL messages since its release in January. Nenad, a co-author of both parts of this series is also the chief architect for WTL and plans to support it and continue to expand it as much as possible. He's already providing support for its use inside Microsoft (although that's not his main job), and has plans for a bug-fix release in the July Platform SDK. Chris, another co-author of both parts, has volunteered web space to maintaining as much of a bug/patch list as we can scrape together.

Is WTL for everyone? No. It's for those folks that are willing to work a little harder to get something a little better. If you've read this far, that probably includes you.

Chris Sells is the Director of Software Engineering at DevelopMentor and the co-author of *Effective COM* and *ATL Internals;* he can be reached http://staff.develop.com/csells.

Dharma Shukla is a Software Design Engineer on the BizTalk Server 2000 group at Microsoft and can be reached at dharmas@microsoft.com.

Nenad Stefanovic is a member of the original ATL/WTL team. He is a Software Development Engineer at Microsoft and can be reached at nenads@microsoft.com.