How to host an IContextMenu, part 11 – Composite extensions – composition

devblogs.microsoft.com/oldnewthing/20041007-00

October 7, 2004

Raymond Chen

Okay, now that we have two context menu handlers we want to compose (namely, the "real" one from the shell namespace and a "fake" one that contains bonus commands we want to add), we can use merge them together by means of a composite context menu handler.

The kernel of the composite context menu is to multiplex multiple context menus onto a single context menu handler, using the menu identifer offsets to route the commands.

Everything else is just typing.

```
class CCompositeContextMenu : public IContextMenu3
{
public:
  // *** IUnknown ***
  STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
  STDMETHODIMP_(ULONG) AddRef();
  STDMETHODIMP_(ULONG) Release();


  // *** IContextMenu ***
  STDMETHODIMP QueryContextMenu(HMENU hmenu,
                          UINT indexMenu, UINT idCmdFirst,
                          UINT idCmdLast, UINT uFlags);
  STDMETHODIMP InvokeCommand(
                          LPCMINVOKECOMMANDINFO lpici);
  STDMETHODIMP GetCommandString(
                          UINT_PTR    idCmd,
                          UINT        uType,
                          UINT      * pwReserved,
                          LPSTR       pszName,
                          UINT        cchMax);


  // *** IContextMenu2 ***
  STDMETHODIMP HandleMenuMsg(
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam);


  // *** IContextMenu3 ***
  STDMETHODIMP HandleMenuMsg2(
                          UINT uMsg,
                          WPARAM wParam,
                          LPARAM lParam,
                          LRESULT* plResult);


  // Constructor
  static HRESULT Create(IContextMenu **rgpcm, UINT cpcm,
                        REFIID riid, void **ppv);


private:


  HRESULT Initialize(IContextMenu **rgpcm, UINT cpcm);
  CCompositeContextMenu() : m_cRef(1), m_rgcmi(NULL), m_ccmi(0) { }
  ~CCompositeContextMenu();
  struct CONTEXTMENUINFO {
    IContextMenu *pcm;
    UINT cids;
  };


  HRESULT ReduceOrdinal(UINT_PTR *pidCmd, CONTEXTMENUINFO **ppcmi);


private:
  ULONG m_cRef;
  CONTEXTMENUINFO *m_rgcmi;
  UINT m_ccmi;
};
```

The local structure `CONTEXTMENUINFO` contains information about each of the context menus that are part of our composite. We need to have the context menu pointer itself, as well as the number of menu identifiers consumed by that context menu by its `IContextMenu::QueryContextMenu` handler. We'll see why as we implement this class.

```
HRESULT CCompositeContextMenu::Initialize(
    IContextMenu **rgpcm, UINT cpcm)
{
  m_rgcmi = new CONTEXTMENUINFO[cpcm];
  if (!m_rgcmi) {
    return E_OUTOFMEMORY;
  }


  m_ccmi = cpcm;
  for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
    CONTEXTMENUINFO *pcmi = &m_rgcmi[icmi];
    pcmi->pcm = rgpcm[icmi];
    pcmi->pcm->AddRef();
    pcmi->cids = 0;
  }


  return S_OK;
}
```

Since a C++ constructor cannot fail, there are various conventions for how one handles failure during construction. One convention, which I use here, is to put the bulk of the work in an `Initialize` method, which can return an appropriate error code if the initialization fails.

(Note that here I am assuming a non-throwing `new` operator.)

Our initialization function allocates a bunch of `CONTEXTMENUINFO` structures and copies the `IContextMenu` pointers (and `AddRef` s them) for safekeeping. (Note that the `m_ccmi` member is not set until after we know that the memory allocation succeeded.)

The destructor therefore undoes these operations.

```
CCompositeContextMenu::~CCompositeContextMenu()
{
  for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
    m_rgcmi[icmi].pcm->Release();
  }
  delete[] m_rgcmi;
}
```

(If you don't understand the significance of the `[]`, here's a refresher.)

The `Create` pattern you saw last time, so this shouldn't be too surprising.

```
HRESULT CCompositeContextMenu::Create(IContextMenu **rgpcm, UINT cpcm,
                                      REFIID riid, void **ppv)
{
  *ppv = NULL;


  HRESULT hr;
  CCompositeContextMenu *self = new CCompositeContextMenu();
  if (self) {
    if (SUCCEEDED(hr = self->Initialize(rgpcm, cpcm)) &&
        SUCCEEDED(hr = self->QueryInterface(riid, ppv))) {
      // success
    }
    self->Release();
  } else {
    hr = E_OUTOFMEMORY;
  }
  return hr;
}
```

And then the standard COM bookkeeping.

```
HRESULT CCompositeContextMenu::QueryInterface(REFIID riid, void **ppv)
{
  IUnknown *punk = NULL;
  if (riid == IID_IUnknown) {
    punk = static_cast<IUnknown*>(this);
  } else if (riid == IID_IContextMenu) {
    punk = static_cast<IContextMenu*>(this);
  } else if (riid == IID_IContextMenu2) {
    punk = static_cast<IContextMenu2*>(this);
  } else if (riid == IID_IContextMenu3) {
    punk = static_cast<IContextMenu3*>(this);
  }


  *ppv = punk;
  if (punk) {
    punk->AddRef();
    return S_OK;
  } else {
    return E_NOINTERFACE;
  }
}


ULONG CCompositeContextMenu::AddRef()
{
  return ++m_cRef;
}


ULONG CCompositeContextMenu::Release()
{
  ULONG cRef = –m_cRef;
  if (cRef == 0) delete this;
  return cRef;
}
```

Now we reach our first interesting method: `IContextMenu::QueryContextMenu` :

```
HRESULT CCompositeContextMenu::QueryContextMenu(
    HMENU hmenu, UINT indexMenu, UINT idCmdFirst,
    UINT idCmdLast, UINT uFlags)
{
  UINT idCmdFirstOrig = idCmdFirst;
  UINT cids = 0;


  for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
    CONTEXTMENUINFO *pcmi = &m_rgcmi[icmi];
    HRESULT hr = pcmi->pcm->QueryContextMenu(hmenu,
                    indexMenu, idCmdFirst, idCmdLast, uFlags);
    if (SUCCEEDED(hr)) {
      pcmi->cids = (USHORT)hr;
      cids += pcmi->cids;
      idCmdFirst += pcmi->cids;
    }
  }


  return MAKE_HRESULT(SEVERITY_SUCCESS, 0, cids);
}
```

We ask each contained context menu in turn to add its commands to the context menu. Here is where you see one of the reasons for the return value of the `IContextMenu::QueryContextMenu` method. By telling tells the container how many menu identifiers you used, the container knows how many are left for others. The container then returns the total number of menu identifiers consumed by all of the context menus.

Another reason for the return value of the `IContextMenu::QueryContextMenu` method is seen in the next helper method:

```
HRESULT CCompositeContextMenu::ReduceOrdinal(
    UINT_PTR *pidCmd, CONTEXTMENUINFO **ppcmi)
{
  for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
    CONTEXTMENUINFO *pcmi = &m_rgcmi[icmi];
    if (*pidCmd < pcmi->cids) {
      *ppcmi = pcmi;
      return S_OK;
    }
    *pidCmd -= pcmi->cids;
  }
  return E_INVALIDARG;
}
```

This method takes a menu offset and figures out which of the contained context menus it belongs to, using the return value from `IContextMenu::QueryContextMenu` to decide how to divide up the identifier space. The `pidCmd` parameter is in/out. On entry, it's the menu offset for the composite context menu; on exit, it’s the menu offset for the contained context menu that is returned via the `ppcmi` parameter.

The `IContextMenu::InvokeCommand` is probably the most complicated, since it needs to support the four different ways of dispatching the command.

```
HRESULT CCompositeContextMenu::InvokeCommand(
                            LPCMINVOKECOMMANDINFO lpici) {


  CMINVOKECOMMANDINFOEX* lpicix =
                reinterpret_cast<CMINVOKECOMMANDINFOEX*>(lpici);
  BOOL fUnicode = lpici->cbSize >= sizeof(CMINVOKECOMMANDINFOEX) &&
                  (lpici->fMask & CMIC_MASK_UNICODE);
  UINT_PTR idCmd = fUnicode ? reinterpret_cast<UINT_PTR>(lpicix->lpVerbW)
                            : reinterpret_cast<UINT_PTR>(lpici->lpVerb);


  if (!IS_INTRESOURCE(idCmd)) {
    for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
      HRESULT hr = m_rgcmi->pcm->InvokeCommand(lpici);
      if (SUCCEEDED(hr)) {
        return hr;
      }
    }
    return E_INVALIDARG;
  }


  CONTEXTMENUINFO *pcmi;
  HRESULT hr = ReduceOrdinal(&idCmd, &pcmi);
  if (FAILED(hr)) {
      return hr;
  }


  LPCWSTR pszVerbWFake;
  LPCWSTR *ppszVerbW = fUnicode ? &lpicix->lpVerbW : &pszVerbWFake;
  LPCSTR pszVerbOrig = lpici->lpVerb;
  LPCWSTR pszVerbWOrig = *ppszVerbW;


  lpici->lpVerb = reinterpret_cast<LPCSTR>(idCmd);
  *ppszVerbW = reinterpret_cast<LPCWSTR>(idCmd);


  hr = pcmi->pcm->InvokeCommand(lpici);


  lpici->lpVerb = pszVerbOrig;
  *ppszVerbW = pszVerbWOrig;


  return hr;
}
```

After some preliminary munging to find the command identifier, we dispatch the invocation in three steps.

First, if the command is being dispatched as a string, then this is the easiest case. We loop through all the contained context menus asking each one if it recognizes the command. Once one does, we are done. And if nobody does, then we shrug and say we don't know either.

Second, if the command being dispatched is an ordinal, we ask `ReduceOrdinal` to figure out which contained context menu handler it belongs to.

Third, we rewrite the `CMINVOKECOMMANDINFO` structure so it is suitable for use by the contained context menu handler. This means changing the `lpVerb` member and possibly the `lpVerbW` member to contain the new menu offset relative to the contained context menu handler rather than being relative to the container. This is complicated slightly by the fact that the Unicode verb `lpVerbW` might not exist. We hide that behind a `pszVerbWFake` local variable which stands in if there is no genuine `lpVerbW`.

Okay, now that you see the basic idea behind distributing the method calls to the appropriate contained context menu, the rest should be comparatively easy.

```
HRESULT CCompositeContextMenu::GetCommandString(
                            UINT_PTR    idCmd,
                            UINT        uType,
                            UINT      * pwReserved,
                            LPSTR       pszName,
                            UINT        cchMax)
{
  HRESULT hr;
  if (!IS_INTRESOURCE(idCmd)) {
    for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
      hr = m_rgcmi[icmi].pcm->GetCommandString(idCmd,
                        uType, pwReserved, pszName, cchMax);
      if (hr == S_OK) {
        return hr;
      }
    }
    if (uType == GCS_VALIDATEA || uType == GCS_VALIDATEW) {
      return S_FALSE;
    }
    return E_INVALIDARG;
  }


  CONTEXTMENUINFO *pcmi;
  if (FAILED(hr = ReduceOrdinal(&idCmd, &pcmi))) {
    return hr;
  }


  return pcmi->pcm->GetCommandString(idCmd, uType,
                        pwReserved, pszName, cchMax);
}
```

The `GetCommandString` method follows the same three-step pattern as `InvokeCommand`.

First, dispatch any string-based commands by calling each contained context menu handler until somebody accepts it. If nobody does, then reject the command. (Note the special handling of `GCS_VALIDATE`, which expects `S_FALSE` rather than an error code.)

Second, if the command is specified by ordinal, ask `ReduceOrdinal` to figure out which contained context menu handler it belongs to.

Third, pass the reduced command to the applicable contained context menu handler.

The last methods are made easier by a little helper function:

```
HRESULT IContextMenu_HandleMenuMsg2(
            IContextMenu *pcm, UINT uMsg, WPARAM wParam,
            LPARAM lParam, LRESULT* plResult)
{
  IContextMenu2 *pcm2;
  IContextMenu3 *pcm3;
  HRESULT hr;
  if (SUCCEEDED(hr = pcm->QueryInterface(
                    IID_IContextMenu3, (void**)&pcm3))) {
    hr = pcm3->HandleMenuMsg2(uMsg, wParam, lParam, plResult);
    pcm3->Release();
  } else if (SUCCEEDED(hr = pcm->QueryInterface(
                    IID_IContextMenu2, (void**)&pcm2))) {
    if (plResult) *plResult = 0;
    hr = pcm2->HandleMenuMsg(uMsg, wParam, lParam);
    pcm2->Release();
  }
  return hr;
}
```

This helper function takes an `IContextMenu` interface pointer and tries to invoke `IContextMenu3::HandleMenuMsg2`; if that fails, then it tries `IContextMenu2::HandleMenuMsg`; and if that also fails, then it gives up.

With this helper function, the last two methods are a piece of cake.

```
HRESULT CCompositeContextMenu::HandleMenuMsg(
            UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  LRESULT lres;   // thrown away
  return HandleMenuMsg2(uMsg, wParam, lParam, &lres);
}
```

The `IContextMenu2::HandleMenuMsg` method is just a forwarder to the `IContextMenu3::HandleMenuMsg2` method:

```
HRESULT CCompositeContextMenu::HandleMenuMsg2(
            UINT uMsg, WPARAM wParam, LPARAM lParam,
            LRESULT* plResult)
{
  for (UINT icmi = 0; icmi < m_ccmi; icmi++) {
    HRESULT hr;
    if (SUCCEEDED(hr = IContextMenu_HandleMenuMsg2(
                    m_rgcmi[icmi].pcm, uMsg, wParam, lParam,
                    plResult))) {
      return hr;
    }
  }
  return E_NOTIMPL;
}
```

And the `IContextMenu3::HandleMenuMsg2` method merely walks through the list of context menu handlers, asking each one whether it wishes to handle the command, stopping when one finally does.

Armed with this composite menu class, we can show it off in our sample program by compositing the "real" context menu with our `CTopContextMenu`, thereby showing how you can combine multiple context menus into one big context menu.

```
HRESULT GetCompositeContextMenuForFile(HWND hwnd,
            LPCWSTR pszPath, REFIID riid, void **ppv)
{
  *ppv = NULL;
  HRESULT hr;


  IContextMenu *rgpcm[2] = { 0 };
  if (SUCCEEDED(hr = GetUIObjectOfFile(hwnd, pszPath,
                        IID_IContextMenu, (void**)&rgpcm[0])) &&
      SUCCEEDED(hr = CTopContextMenu::Create(
                        IID_IContextMenu, (void**)&rgpcm[1])) &&
      SUCCEEDED(hr = CCompositeContextMenu::Create(rgpcm, 2, riid, ppv))) {
      // yay
  }
  if (rgpcm[0]) rgpcm[0]->Release();
  if (rgpcm[1]) rgpcm[1]->Release();


  return hr;
}
```

This function builds the composite by creating the two contained context menu handlers, then creating a composite context menu that contains both of them. We can use this function by making the same one-line tweak to the `OnContextMenu` function that we tweaked last time:

```
void OnContextMenu(HWND hwnd, HWND hwndContext, int xPos, int yPos)
{
  POINT pt = { xPos, yPos };
  if (pt.x == -1 && pt.y == -1) {
    pt.x = pt.y = 0;
    ClientToScreen(hwnd, &pt);
  }


  IContextMenu *pcm;
  if (SUCCEEDED(GetCompositeContextMenuForFile(
                  hwnd, L”C:\\Windows\\clock.avi”,
                  IID_IContextMenu, (void**)&pcm))) {
    …
```

Notice that with this composite context menu, the menu help text that we update in our window title tracks across both the original file context menu and our “Top” context menu. Commands from either half are also invoked successfully.

The value of this approach over the method from part 9 is that you no longer have to coordinate the customization of the context menu between two pieces of code. Under the previous technique, you had to make sure that the code that updated the menu help text was in sync with the code that added the custom commands.

Under the new method, all the customizations are kept in one place (in the “Top” context menu which is inside the composite context menu), so that the window procedure doesn’t need to know what customizations have taken place. This becomes more valuable if there are multiple points at which context menus are displayed, some uncustomized, others customized in different ways. Centralizing the knowledge of the customizations simplifies the design.

Okay, I think that’s enough on context menus for now. I hope you’ve gotten a better understanding of how they work, how you can exploit them, and most importantly, how you can perform meta-operations on them with techniques like composition.

There are still some other things you can do with context menus, but I’m going to leave you to experiment with them on your own. For example, you can use the IContextMenu::GetCommandString method to walk the menu and obtain a language-independent command mame for each item. This is handy if you want to, say, remove the “delete” option: You can look for the command whose language-independent name is “delete”. This name does not change when the user changes languages; it will always be in English.

As we’ve noticed before, you need to be aware that many context menu handlers don’t implement the IContextMenu::GetCommandString method fully, so there will likely be commands that you simply cannot get a name for. Them’s the breaks.

[Editing errors corrected, 11am.]

Raymond Chen

Follow