Wouldn't the Recycle Bin sample program have been simpler without COM?

devblogs.microsoft.com/oldnewthing/20131220-00

December 20, 2013

Raymond Chen

Steve Wolf suggests that the sample program would have been much simpler had the shell extension model been a flat Win32 interface.

Okay, let's try it.

Since this is an extension model, each extension needs to specify the callbacks for each namespace operation. Perhaps it could have been done like this:

```
HRESULT (CALLBACK *SHELLFOLDER_EXTENDHANDLER)(
    void *lpContext,
    OBJECTTYPE type, void **phObject);
HRESULT (CALLBACK *SHELLFOLDER_PARSEDISPLAYNAMEHANDLER)(
    void *lpContext,
    HWND hwnd, LPBINDCTX pbc, LPWSTR pszDisplayName,
    ULONG *pchEaten, PIDLIST_RELATIVE *ppidl, ULONG *pdwAttributes);
HRESULT (CALLBACK *SHELLFOLDER_ENUMOBJECTSHANDLER)(
    void *lpContext,
    HWND hwnd, SHCONTF grfFlags, HENUMIDLIST *pheidl);
HRESULT (CALLBACK *SHELLFOLDER_BINDTOOBJECTHANDLER)(
    void *lpContext,
    PCUIDLIST_RELATIVE pidl, LPBINDCTX pbc,
    OBJECTTYPE type, void **phObject);
HRESULT (CALLBACK *SHELLFOLDER_BINDTOSTRAGEHANDLER)(
    void *lpContext,
    PCUIDLIST_RELATIVE pidl, LPBINDCTX pbc,
    OBJECTTYPE type, void **phObject);
HRESULT (CALLBACK *SHELLFOLDER_COMPAREIDSHANDLER)(
    void *lpContext,
    LPARAM lParam, PCUIDLIST_RELATIVE pidl1,
    PCUIDLIST_RELATIVE pidl2);
... (etc) ...
HFOLDER CreateShellFolderImplementation(
    SHELLFOLDER_EXTENDHANDLER pfnExtend,
    SHELLFOLDER_PARSEDISPLAYNAMEHANDLER pfnParseDisplayName,
    SHELLFOLDER_ENUMOBJECTSHANDLER pfnEnumObjects,
    SHELLFOLDER_BINDTOOBJECTHANDLER pfnBindToObject,
    SHELLFOLDER_BINDTOSTRAGEHANDLER pfnBindToStorage,
    SHELLFOLDER_COMPAREIDSHANDLER pfnCompareIDs,
    ... (etc) ...
    void *lpContext);
```

This would be the function that allows a third party to create a shell folder implementation. You pass it a bunch of flat callback functions, one for each operation that a shell folder supports, so that when the application tries to perform that operation on your custom folder, the operating system can ask your custom implementation to do that thing.

If additional shell folder operations are added in the future, the operating system needs to know how to ask your shell extension whether it knows how to do those extended things. That's what the `Extend` method is for. The operating system could ask to extend your object to one that supports `HFOLDER2` operations.

Actually, if you look at it, these are exactly the same as COM methods. The first parameter says what object you are operating on ("`this`"), and the rest are the parameters.

Okay, so I'm setting up a straw man that looks just like COM. So let's do something that looks very different from COM. We could use the window procedure paradigm:

```
HRESULT (CALLBACK *SHELLFOLDER_INVOKE)(
    void *lpContext,
    FOLDERCOMMAND cmd, void *parameters);
HFOLDER CreateShellFolderImplementation(
    SHELLFOLDER_INVOKE pfnInvoke,
    void *lpContext);
```

Your invoke function receives a `FOLDERCOMMAND` enumeration which specifies what command the client is trying to perform, and then switches on the command to perform the command, or returns `E_NOTIMPL` if you don't handle the command. Since each of the methods takes different parameters, we have to do some work to pack them up into a generic parameter block, and then unpack it on the receiving end. Let's assume some helper functions that do this packing and unpacking.

```
HRESULT UnpackParseDisplayName(
    void *parameters,
    HWND *phwnd,
    LPBINDCTX *ppbc,
    LPWSTR *ppszDisplayName,
    ULONG **ppchEaten,
    PIDLIST_RELATIVE **ppidl,
    ULONG **ppdwAttributes);
);
HRESULT UnpackEnumObjects(
    void *parameters,
    HWND *phwnd,
    SHCONTF *pgrfFlags,
    HENUMIDLIST **ppheidl);
HRESULT AwesomeShellFolderInvoke(
    void *lpContext,
    FOLDERCOMMAND cmd,
    void *parameters)
{
  HRESULT hr = E_NOTIMPL;
  CAwesome *self = reinterpret_cast<CAwesome*>(lpContext);
  switch (cmd) {
  case FOLDERCOMMAND_PARSEDISPLAYNAME:
    {
      HWND hwnd;
      LPBINDCTX pbc;
      LPWSTR pszDisplayName;
      ULONG *ppchEaten;
      PIDLIST_RELATIVE *pidl;
      ULONG *pdwAttributes;
      hr = UnpackParseDisplayName(parameters, &hwnd, &pbc,
              &pszDisplayName, &ppchEaten, &pidl,
              &pdwAttributes);
      if (SUCCEEDED(hr)) {
        hr = ... do the actual work ...
      }
    }
    break;
  case FOLDERCOMMAND_ENUMOBJECTS:
    {
      HWND hwnd;
      SHCONTF grfFlags;
      HENUMIDLIST *pheidl;
      hr = UnpackEnumObjects(parameters, &hwnd, &grfFlags,
              &pheidl);
      if (SUCCEEDED(hr)) {
        hr = ... do the actual work ...
      }
    }
    break;
    ... (etc) ...
  }
```

```
  return hr;
}
```

This could be made a lot simpler with the addition of some helper functions.

```
HRESULT DispatchParseDisplayName(
  HRESULT (CALLBACK *)(
    void *lpContext,
    HWND hwnd, LPBINDCTX pbc, LPWSTR pszDisplayName,
    ULONG *pchEaten, PIDLIST_RELATIVE *ppidl, ULONG *pdwAttributes),
  void *lpContext,
  void *parameters);
HRSEULT DispatchEnumObjects(
  HRESULT (CALLBACK *)(
    void *lpContext,
    HWND hwnd, SHCONTF grfFlags, HENUMIDLIST *pheidl),
  void *lpContext,
  void *parameters);
```

The implementation would then go like this:

```
HRESULT AwesomeParseDisplayName(
    void *lpContext,
    HWND hwnd, LPBINDCTX pbc, LPWSTR pszDisplayName,
    ULONG *pchEaten, PIDLIST_RELATIVE *ppidl, ULONG *pdwAttributes)
{
  CAwesome *self = reinterpret_cast<CAwesome*>(lpContext);
  HRESULT hr;
  ... do the actual work ...
  return hr;
}
HRESULT AwesomeEnumObjects(
    void *lpContext,
    HWND hwnd, SHCONTF grfFlags, HENUMIDLIST *pheidl),
{
  CAwesome *self = reinterpret_cast<CAwesome*>(lpContext);
  HRESULT hr;
  ... do the actual work ...
  return hr;
}
HRESULT AwesomeShellFolderInvoke(
    void *lpContext,
    FOLDERCOMMAND cmd,
    void *parameters)
{
  switch (cmd) {
  case FOLDERCOMMAND_PARSEDISPLAYNAME:
    return DispatchParseDisplayName(AwesomeParseDisplayName,
              lpContext, parameters);
  case FOLDERCOMMAND_ENUMOBJECTS:
    return DispatchEnumObjects(AwesomeEnumObjects,
              lpContext, parameters);
    ... (etc) ...
  }
  return E_NOTIMPL;
}
```

You might decide to make the parameter packing transparent instead of opaque, so that they are passed as, say, an array of generic types like `VARIANT`s. (Note that I'm abusing `VARIANT`s here. These are not valid `VARIANT`s, but it saves me from having to declare my own generic type. This is just a design discussion, not an actual implementation.)

```
HRESULT (CALLBACK *SHELLFOLDER_INVOKE)(
    void *lpContext,
    FOLDERCOMMAND cmd,
    VARIANT *rgvarArgs,
    UINT cArgs);
// error checking elided for expository purposes
// In real life, you would have to validate cArgs
// and the variant types.
HRESULT AwesomeShellFolderInvoke(
    void *lpContext,
    FOLDERCOMMAND cmd,
    VARIANT *rgvarArgs,
    UINT cArgs)
{
  CAwesome *self = reinterpret_cast<CAwesome*>(lpContext);
  switch (cmd) {
  case FOLDERCOMMAND_PARSEDISPLAYNAME:
    return self->ParseDisplayName(
      reinterpret_cast<HWND>(rgvarArgs[0]->byref),
      reinterpret_cast<LPBINDCTX>(rgvarArgs[1]->byref),
      reinterpret_cast<LPWSTR>(rgvarArgs[2]->byref),
      reinterpret_cast<ULONG*>(rgvarArgs[3]->byref),
      reinterpret_cast<PIDLIST_RELATIVE*>(rgvarArgs[4]->byref),
      reinterpret_cast<ULONG**>(rgvarArgs[5]->byref));
  case FOLDERCOMMAND_ENUMOBJECTS:
    return self->EnumObjects(
      reinterpret_cast<HWND>(rgvarArgs[0]->byref),
      reinterpret_cast<SHCONTF>(rgvarArgs[1]->lVal),
      reinterpret_cast<HENUMIDLIST *>(rgvarArgs[2]->byref));
    ... (etc) ...
  }
  return E_NOTIMPL;
}
```

(This is basically the plug-in model that some people have chosen to pursue. It is also basically the same as `IDispatch::Invoke`.)

Okay, that's how you implement the plug-in. Now how do you call it?

You would have to pack the parameters, then call through the `Invoke` method with your command ID. For example, a call to `FOLDERCOMMAND_ENUMOBJECTS` would go like this:

```
// was: hr = psf->EnumObjects(hwnd, shcontf, &peidl);
// now:
HENUMIDLIST heidl;
VARIANT args[3];
args[0].vt = VT_BYREF;
args[0].byref = hwnd;
args[1].vt = VT_I4;
args[1].lVal = shcontf;
args[2].vt = VT_BYREF;
args[2].byref = &heidl;
hr = InvokeShellFolder(hsf, FOLDERCOMMAND_ENUMOBJECTS, args, 3);
```

Yuck.

Let's assume that the shell provides helper functions that do all this parameter packing for you. (This is more than certain plug-in models give you.)

```
HRESULT ShellFolder_ParseDisplayName(
    HSHELLFOLDER hsf,
    HWND hwnd, LPBINDCTX pbc, LPWSTR pszDisplayName,
    ULONG *pchEaten, PIDLIST_RELATIVE *ppidl, ULONG *pdwAttributes)
{
  VARIANT args[6];
  args[0].vt = VT_BYREF;
  args[0].byref = hwnd;
  args[1].vt = VT_BYREF;
  args[1].byref = pbc;
  args[2].vt = VT_BYREF;
  args[2].byref = pszDisplayName;
  args[3].vt = VT_BYREF;
  args[3].byref = pchEaten;
  args[4].vt = VT_BYREF;
  args[4].byref = ppidl;
  args[5].vt = VT_BYREF;
  args[5].byref = pdwAttributes;
  return InvokeShellFolder(hsf, FOLDERCOMMAND_PARSEDISPLAYNAME,
                           args, 6);
}
HRESULT ShellFolder_EnumObjects(
    HSHELLFOLDER hsf,
    HWND hwnd, SHCONTF grfFlags, HENUMIDLIST *pheidl)
{
  VARIANT args[3];
  args[0].vt = VT_BYREF;
  args[0].byref = hwnd;
  args[1].vt = VT_I4;
  args[1].lVal = shcontf;
  args[2].vt = VT_BYREF;
  args[2].byref = &heidl;
  return InvokeShellFolder(hsf, FOLDERCOMMAND_ENUMOBJECTS, args, 3);
}
... (etc) ...
```

The naming convention above is kind of awkward, so let's give them a bit less clumsy names.

```
HRESULT ParseShellFolderDisplayName(
    HSHELLFOLDER hsf,
    HWND hwnd, LPBINDCTX pbc, LPWSTR pszDisplayName,
    ULONG *pchEaten, PIDLIST_RELATIVE *ppidl, ULONG *pdwAttributes);
HRESULT EnumShellFolderObjects(
    HSHELLFOLDER hsf,
    HWND hwnd, SHCONTF grfFlags, HENUMIDLIST *pheidl);
... (etc) ...
```

Okay, now that we have a flat API, let's convert the original code. The first function now goes like this:

```
HRESULT BindToCsidl(int csidl,
    // REFIID riid, void **ppv
    HSHELLFOLDER *phsf)
{
 HRESULT hr;
 PIDLIST_ABSOLUTE pidl;
 hr = SHGetSpecialFolderLocation(NULL, csidl, &pidl);
 if (SUCCEEDED(hr)) {
  // IShellFolder *psfDesktop;
  HSHELLFOLDER hsfDesktop;
  hr = SHGetDesktopFolder(&hsfDesktop);
  if (SUCCEEDED(hr)) {
   if (pidl->mkid.cb) {
    // hr = psfDesktop->BindToObject(pidl, NULL, riid, ppv);
    hr = BindToShellFolderObject(hsfDesktop, pidl, NULL, phsf);
   } else {
    // hr = psfDesktop->QueryInterface(riid, ppv);
    *phsf = hsfDesktop;
    hsfDesktop = nullptr; // transfer to owner
    hr = S_OK;
   }
   // psfDesktop->Release();
   if (hsfDesktop) ShellFolder_Destroy(hsfDesktop);
  }
  CoTaskMemFree(pidl);
 }
 return hr;
}
```

What happened here? The `IShellFolder` interface was replaced by a `HSHELLFOLDER` flat handle. Flat APIs use handles to refer to objects instead of interface pointers.

A method call on an interface pointer becomes a flat API call. In general, `pInterface->VerbNoun(args)` gets flattened to `VerbInterfaceNoun(h, args)`. But that's just renaming and doesn't change the underlying complexity of the issue.

I could've added reference counting to these flat objects, but then I would be accused of intentionally making it look like COM, so let's say that these flat objects are not reference-counted. Therefore, we have to be more careful about not destroying the object we plan on returning.

On to the next two functions:

```
void PrintDisplayName(
    // ~~IShellFolder *psf,~~
    HSHELLFOLDER hsf,
    PCUITEMID_CHILD pidl, SHGDNF uFlags, PCTSTR pszLabel)
{
 STRRET sr;
 // HRESULT hr = ~~psf->GetDisplayNameOf(~~pidl, uFlags, &sr);
 HRESULT hr = GetShellFolderDisplayNameOf(hsf, pidl, uFlags, &sr);
 if (SUCCEEDED(hr)) {
  PTSTR pszName;
  hr = StrRetToStr(&sr, pidl, &pszName);
  if (SUCCEEDED(hr)) {
   _tprintf(TEXT("%s = %s\n"), pszLabel, pszName);
   CoTaskMemFree(pszName);
  }
 }
}
void PrintDetail(
    // ~~IShellFolder2 *psf,~~
    HSHELLFOLDER hsf,
    PCUITEMID_CHILD pidl,
    const SHCOLUMNID *pscid, PCTSTR pszLabel)
{
 VARIANT vt;
 // HRESULT hr = ~~psf->GetDetailsEx(~~pidl, pscid, &vt);
 HRESULT hr = GetShellFolderDetailsEx(hsf, pidl, pscid, &vt);
 if (SUCCEEDED(hr)) {
  hr = VariantChangeType(&vt, &vt, 0, VT_BSTR);
  if (SUCCEEDED(hr)) {
   _tprintf(TEXT("%s: %ws\n"), pszLabel, V_BSTR(&vt));
  }
  VariantClear(&vt);
 }
}
```

Not really all that different. Last function:

```
int __cdecl _tmain(int argc, PTSTR *argv)
{
 HRESULT hr = CoInitialize(NULL);
 if (SUCCEEDED(hr)) {
  // IShellFolder2 *psfRecycleBin;
  HSHELLFOLDER hsfRecycleBin;
  hr = BindToCsidl(CSIDL_BITBUCKET, &hsfRecycleBin);
  if (SUCCEEDED(hr)) {
   // IEnumIDList *peidl;
   HENUMIDLIST heidl;
   // hr = psfRecycleBin->EnumObjects(NULL,
   hr = EnumShellFolderObjects(hsfRecycleBin, NULL,
     SHCONTF_FOLDERS | SHCONTF_NONFOLDERS, &heidl);
   if (hr == S_OK) {
    PITEMID_CHILD pidlItem;
    // while (peidl->Next(1, &pidlItem, NULL) == S_OK) {
    while (EnumerateNextShellFolderObject(heidl, 1, &pidlItem, NULL) == S_OK) {
     _tprintf(TEXT("------------------\n"));
     PrintDisplayName(hsfRecycleBin, pidlItem,
                      SHGDN_INFOLDER, TEXT("InFolder"));
     PrintDisplayName(hsfRecycleBin, pidlItem,
                      SHGDN_NORMAL, TEXT("Normal"));
     PrintDisplayName(hsfRecycleBin, pidlItem,
                      SHGDN_FORPARSING, TEXT("ForParsing"));
     PrintDetail(hsfRecycleBin, pidlItem,
                 &SCID_OriginalLocation, TEXT("Original Location"));
     PrintDetail(hsfRecycleBin, pidlItem,
                 &SCID_DateDeleted, TEXT("Date deleted"));
     PrintDetail(hsfRecycleBin, pidlItem,
                 &PKEY_Size, TEXT("Size"));
     CoTaskMemFree(pidlItem);
    }
   }
   // psfRecycleBin->Release();
   DestroyShellFolder(hsfRecycleBin);
  }
  CoUninitialize();
 }
 return 0;
}
```

So we see that flattening the API didn't really change the code at all. You're still invoking methods on objects. Whether you use a flat API to do it or an object-based API is just changing the decorations. The underlying logic doesn't change.

One disadvantage of the flat version is that it requires everything to be mediated by the shell. Instead of invoking a method directly on the object, you have to call the flat function in the shell, which then packages up the call and dispatches it, and the recipient then needs to unpack the parameters (possibly with help from the shell) before finally getting around to doing the actual work.

It also means that any interface change requires an operating system upgrade, since the mediator (the shell) needs to understand the new interface.

But if this whole object-oriented syntax really annoys you and you want a flat API, then feel free to add the line

```
#define CINTERFACE
```

before including COM header files. If you do that, then you get the old flat C-style version of COM. Instead of the `p->Method(args)` new hotness, you can stick to the old trustworthy `p->lpVtbl->Method(p, args)` version, or use the `InterfaceName_MethodName(p, args)` helper macro.

Raymond Chen

Follow