# Workplace Shell Programming with DTS C++

**Scott Danforth**

**(To appear in the August 96 issue of OS/2 Magazine)**

## Introduction

DirectToSOM C++ is based on the following simple principle: when you define a C++ class that descends from SOMObject, the compiler implements it using SOM. This let's you use all of C++ to define and use SOM classes, which is great news for programmers of the Workplace Shell and other SOM-based frameworks.

But, as is often the case with simple ideas, the devil is in the details. C++ is not all that simple in the first place. And, its object model is somewhat different from SOM's. So, although DTS C++ makes it easier than ever before to define SOM classes, you've got to expect some pretty interesting details. If you want to use DTS C++ to do WPS programming, you'll be more successful (and less frustrated by problems when they arise) if you understand some of these details.

The problems you'll run into mostly relate to the current (evolving) status of DTS C++ compilers and supporting tools. Also, some modification of WPS IDL files is required. To help out, let me give you my perspective on some of these things and illustrate the resulting ideas with a simple WPS class. I'll assume that you're generally familiar with SOM and WPS programming.

## Can your Object Model do This?

SOM was chosen to implement the WPS for a number of reasons. These include SOM's neutrality with respect to languages and compilers (any language that can call external procedures can use SOM), and support for release-to-release binary compatability. DTS C++ classes gain these benefits while offering SOM programmers enhanced performance and new capabilities. (For example, you can now have SOM objects as local variables on your stack, with full constructor/destructor support.)

Another important reason for using SOM is that it supports the WPS programming model, by which I mean a particular set of intentions for how programmers should use subclassing and class replacement to inherit and specialize the behavior of WPS objects. But, the WPS programming model relies on aspects of SOM that are not represented by corresponding C++ capabilities. What does a DTS C++ programmer do about these things?

For example, the WPS programming model requires the ability to define metaclasses and to perform parent method calls from overrides. Yet, the C++ object model includes neither metaclasses nor parent method calls. How do you access these non-C++ capabilities when programming in DTS C++ ?

### Metaclasses

SOM metaclasses are supported by a DTS C++ pragma. For example, the following DTS C++ header file illustrates use of this pragma on line 6 to define a subclass of WPFolder with a corresponding metaclass.

```
1   #include <wpfolder.hh>
2   struct M_MyFolder : M_WPFolder {
```

```
// new class methods, variables, and overrides...
};
struct MyFolder : WPFolder {
#pragma SOM_Metaclass(M_MyFolder)
// new instance methods, variables, and overrides...
}
```

In DTS C++, any struct or class that derives from a SOM class is also a SOM class. Thus, M_MyFolder and MyFolder in this example are SOM classes because M_WPFolder and WP_Folder are SOM classes (ultimately, because they descend from SOMObject). The file wpfolder.hh and those that it includes provide this information to the DTS C++ compiler.

DTS C++ header files can be written by DTS C++ programmers, or can be produced from IDL by the SOMobjects Toolkit hh emitter. For example, a SOMobjects compiler command to produce wpfolder.hh from wpfolder.idl is:

```
sc -shh -mnoqualifytypes -musehpass -S1000000 wpfolder.idl
```

where wpfolder.idl is the interface definition for WPFolder provided by the Developer's Toolkit that comes with Visual Age C++. The reason for the `-mnoqualifytypes` switch is that the hh emitter uses nested C++ classes to implement name scoping corresponding to modules in IDL. Normally, the SOM compiler prepends module names to interface names (to create what are called C-scoped names), but the switch prevents this, as required to support the desired mapping approach. The `-musehpass` means that the emitter should emit C_h passthru statements if no C_hh passthru statements are found. This is important because much of the information required by WPS programs is located in C_h passthru statements. The `-S` switch increases the string table size.

### Parent Method Calls

Parent method calls are an important part of the WPS programming model. WPS classes introduce various "hook" methods that are invoked when handling desktop events, and subclasses tie into the overall operation of desktop objects by overriding these methods. Overrides participate in the overall handling of an event and then make a parent method call so ancestor classes can participate too.

The semantics of a parent method call is based on the actual runtime class hierarchy. You can think of a parent call as starting from the class whose override is executing and then searching up the runtime class hierarchy for the next definition of the method. In single inheritance SOM class hierarchies such as the WPS, the function somParentResolve can be used to make parent method calls.

The closest concept in C++ is a "nonvirtual" method call, which is denoted with the following syntax:

```
obj->X::foo(...)
```

where X is any class that inherits the virtual function foo. But, this syntax is not restricted to overrides, and there is no concept of starting from a "current" class and searching upwards in the class hierarchy for the next definition. Instead, X's implementation for foo is used. In SOM, this is done using the function `somClassResolve`, and this is different from a parent method call. The reason for the difference concerns class replacement, a powerful and important feature of the WPS programming model. Let's review how this works.

As illustrated in Figure 1, different user-defined classes can decide to specialize the behavior of a

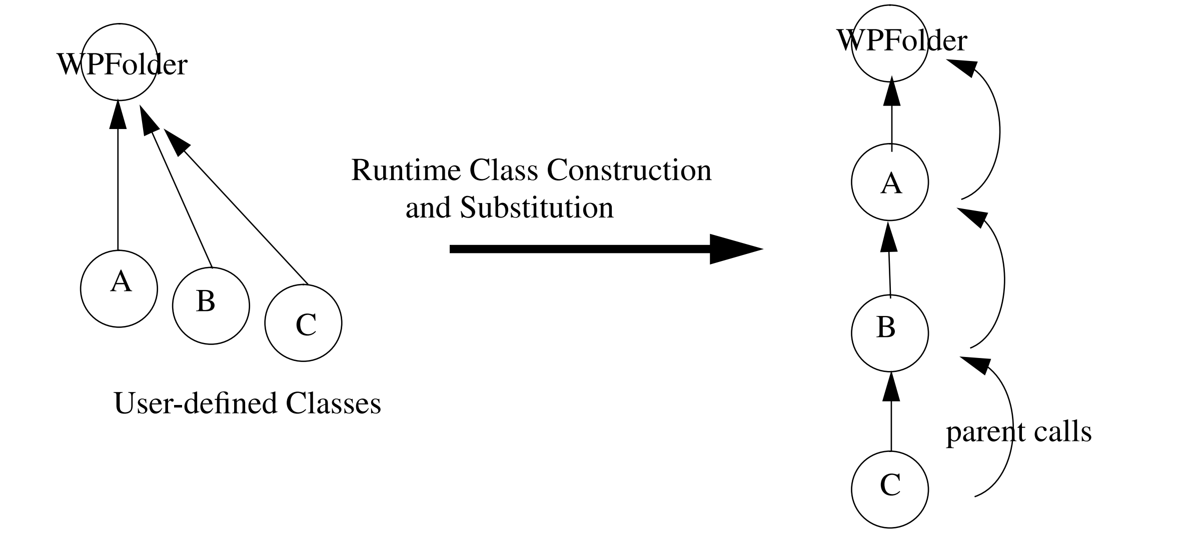

Figure 1. Class Replacement and Parent Method Calls

WPS framework class in different ways (e.g., to add different items to the folder popup menu). In this example, after class A is built it “replaces” WPFolder. After this, if the WPS creates folder objects on the desktop, they will actually be instances of A. Also, when B is built as a subclass of WPFolder it actually receives A as its parent class (instead of WPFolder). Then, in this example, B replaces WPFolder, so that when C is built, its parent is actually B. After all these classes are built, correct functioning of desktop folders (which will be instances of C) requires the parent calls diagrammed in Figure 1. This gives each class’s override a chance to participate in folder events.

But, nonvirtual method calls to WPFolder will simply call WPFolder’s code. So, if C used a non-virtual method call to invoke it’s static parent’s code (remember, C doesn’t know about A or B), this would call WPFolder’s code and skip B and A’s overrides. This is why parent method calls are so important in the WPS programming model.

Currently available DTS C++ compilers don’t provide special support for making SOM parent method calls. This should change in the near future, but in the mean time it’s necessary to perform parent method calls by using the SOM API directly. Luckily, the hh emitter provides the necessary support. This is done with various macro and data structure definitions.

## An Example

This example defines a WPFolder subclass that adds a popup menu option that responds with a DosBeep. This keeps the WPS code simple, so we can focus on the DTS C++ details.

### Start with an IDL declaration of the new Class

In general, the DTS C++ approach to developing a new WPS class is to describe that class in IDL, and then use the hh and hc emitters to generate the header file for the class and a DTS C++ implementation skeleton file (also, sometimes called an implementation template file). Because the header file for any class needs to include the headers for the class’s parents, you ultimately need the headers for all the ancestors of your new class. In general, it is best that all these headers be generated with the same code level of the header emitter. This is especially important in the case of DTS C++ headers, simply because the emitted output is evolving. Old headers are likely to contain various defects that are not present in more recently emitted headers.

Theoretically, there is no need to use IDL when defining subclasses of WPS framework classes. You should be able to start with a .hh file that you write yourself. Currently, however, IDL is useful because the hh emitter (which takes IDL as input) includes parent method call support in the hh file that it produces. Also, there are a number of important DTS C++ pragmas that you would have to use in your hand-written headers. If you use the hh emitter, you don’t have to deal with any of these details. WPS programmers are already used to starting with IDL, so it is nice that using DTS C++ doesn’t preclude this approach. The same overall development process used for implementing a SOM class using the C bindings can now be used with DTS C++.

Here is the IDL for the this example:

```
1   // file: ex1.idl
2   #include <wpfolder.idl>
3   iinterface EX1: WPFolder {
4       implementation {
5           dtsdefaults;
6           majorversion=1; minorversion=1;
7           wpModifyPopupMenu: override; //   Add option to popup menu
8           wpMenuItemSelected: override; //  Process menu selections
9           passthru C_hh =“”
10      “   #define INCL_WIN”
11      “   #define INCL_DOS”
12      “   #define INCL_GPIBITMAPS”
13      “   #define INCL_DOSERRORS”
14      “   #include <os2.h>”
15      “   #define INCL_WPFOLDER”
16      “   #include <pmwp.h>”
17      “   #define ID_EX1MENU    (WPMENUID_USER+1)// menu option”
18      “   #define ID_DOEX1THING (WPMENUID_USER+2)// option selection”;
19       };
20  }
```

The important modifier here is dtsdefaults (which tells the emitter you are using DTS C++ to implement the class, and that you want to let the DTS C++ compiler define copy constructors and assignment oeprators for you -- the hh and hc emitters work together to make this happen, based on an understanding of when the DTS C++ compiler will generate implicit definitions for these things). If you forget to use dtsdefaults, it isn’t serious, but you’ll have additional code to define in your implementation skeleton. Also, there’s a C_hh passthru, so -musepassh won’t be needed when generating the header file.

### Produce the necessary DTS C++ implementation skeleton and header files

We can begin creating the needed .hh headers with the following command.

```
sc -shh -mnoqualifytypes -S1000000 ex1.idl
```

But, here’s where we start running into problems. As you can see, line 2 of ex1.idl includes the IDL for WPFolder. And, WPFolder’s IDL includes the IDL for it’s parent, and etc. As a result, compiling ex1.idl ends up pulling in a number of WPS IDL files. And there are some problems with this IDL (as it appears in the VAC++ bundle, which is what I’m using for this example) that need to be fixed. For example, the int on line 23 of wptypes.idl needs to be changed to long, and the overrides of somInit and somUninit on line 857 of wpobject.idl need to be deleted.

Wptypes.idl is interesting. Its only purpose is to give the SOM compiler a consistent picture of the data types used in WPS interfaces. A header corresponding to this IDL is never actually included into any implementation code. Instead, the actual data types and definitions used for WPS programming are contained in the header files os2.h and pmwp.h (e.g., as made available by the above C_hh passthru.), and in IDL passthru statements.

As for wpobject.idl, it makes no sense to override both somInit and also somDefaultInit, and the new chk emitter used by the SOMobjects Toolkit flags this error. At runtime, the somDefaultInit method will be chosen, so it is correct to delete somInit. To avoid semantic checks, you can invoke the SOM compiler using the -mnochk switch, but this is not recommended.

Once the above problems have been corrected and ex1.hh has been created, we need the hh files for EX1's ancestors (because these will be included into an overall compiled program). These ancestors are WPFolder, WPFileSystem, WPObject, and SOMObject. The hh emitter is used to create the corresponding hh header files, but remember that WPS IDL doesn't include specially designed C_hh passthrus. Actually, it should be possible to create self-contained IDL for the WPS, which would get rid of the need for passthru statements entirely. But in the mean time, we need to deal with the current IDL, which provides C_h passthru statements.

Using -musehpass when emitting WPS header files helps, but this isn't quite enough. The C_h passthru on line 255 of wpobject.idl needs to be changed to C_h_after. Also, it is necessary to delete a few C language macro definitions that aren't appropriate for DTS C++. These are on lines 259-264 and lines 816-819 of wpobject.idl (these definitions are also unnecessary for the C bindings). The macro definition on lines 267-272 raises questions in my mind, but doesn't influence this example. After the above changes are made, the following commands produce the necessary headers

```
set hh=sc -shh -mnoqualifytypes
%hh% somobj.idl somcls.idl somcm.idl
%hh% -musehpass -S1000000 wpfolder.idl wpflsys.idl wpobject.idl
```

We generate headers for the SOM kernel classes because we want all bindings included by our program to correspond to the same emitter level. The desirability of this was mentioned earlier. To help you verify emitter output, version numbers are provided at the top of emitted hh files.

The hh emitter is in the file emithh.dll, which is located by libpath (and the beginlibpath environment variable). The efw file is cpp.efw, a text file that defines output patterns used by the emitter. It is located by the sminclude environment variable. Minor changes to hh emitter output can be made by editing this file. The DTS C++ implementation template emitter is in the file emitdtm.dll and is supported by the efw file dtm.efw.

### Write the DTS C++ Implementation

Once the necessary .hh files have been created, we can define an implementation for EX1. The first step of this process is to use the hc emitter to create an implementation template file from IDL, as follows:

```
sc -shc ex1.idl
```

This produces the following file:

```
// Generated from ex1.idl at 04/18/96 12:24:48 EDT
```

```
// By IBM DTS C++ implementation template emitter version 1.2
// Using hc.efw file version 1.4
#include <ex1.hh>

/*  Begin implementation of interface EX1  */

// default ctor for dtsdefaults
EX1::EX1() { }

//   Add option to the popup menu
::BOOL EX1::wpModifyPopupMenu(  ::HWND hwndMenu,
                                ::HWND hwndCnr,
                                ::ULONG iPosition)
{ }

//   Process input from the added menu option.
::BOOL EX1::wpMenuItemSelected(  ::HWND hwndFrame,
                                 ::ULONG ulMenuId)
{ }
```

The default constructor can be left empty in this case, since EX1 introduces no instance data whose initialization is required. So all we have to do is fill in the two member function definitions. The result is shown below.

```
// Generated from ex1.idl at 04/18/96 12:24:48 EDT
// By IBM DTS C++ implementation template emitter version 1.2
// Using hc.efw file version 1.4 #include <ex1.hh>

/*  Begin implementation of interface EX1  */

// default ctor for dtsdefaults
EX1::EX1()
 { }

/*  this local fcn returns the module handle for the EX1 dll  */
static HMODULE hmod = 0;
HMODULE gethmod()
{
    if (!hmod) {
        string path  = SOMClassMgrObject->somLocateClassFile(
                                        somIdFromString("EX1"),0,0);
        DosQueryModuleHandle(path,&hmod);
    }
    return hmod;
}

//   Add option to the popup menu
::BOOL EX1::wpModifyPopupMenu(  ::HWND hwndMenu,
                                ::HWND hwndCnr,
                                ::ULONG iPosition)
```

```
{
    wpInsertPopupMenuItems(hwndMenu, iPosition, gethmod(), ID_EX1MENU, 0);
    return (SOM_PARENT_EX1(this,
                            wpModifyPopupMenu,
                            WPObject,
                            SOM_MTOKEN_WPObject(wpModifyPopupMenu))
                (this, hwndMenu, hwndCnr, iPosition));
}

//  Process input from the added menu option.
::BOOL EX1::wpMenuItemSelected(::HWND hwndFrame, ::ULONG ulMenuId)
{
    if (ulMenuId == ID_DOEX1THING) {
        DosBeep(1000,200);
        DosBeep(1000,200);
        DosBeep(1000,200);
        DosBeep(800,400);
        return 1;
     }
    else   return (SOM_PARENT_EX1(this,
                                   wpMenuItemSelected,
                                   WPObject,
                                   SOM_MTOKEN_WPObject(wpMenuItemSelected))
                   (this, hwndFrame, ulMenuId));
}
```

SOM_PARENT_EX1 is a macro provided by ex1.hh that supports parent method calls from any EX1 method override. It takes four arguments: the target object, the method name, the class that introduces the method, and the method token. This evaluates to a pointer to the parent's method procedure, which is applied to the method call arguments. The hardest part of using the parent call macro is knowing the introducing class for a method (this information is also required to choose the name of the method token macro). In the above example, both wpModifyPopupMenu and wpMenuItemSelected are introduced by WPObject.

Obviously, compiler support for parent method calls will be a great improvement. Then the compiler is responsible for determining the introducing class. Anticipating one possible syntax, a parent method call might be coded in DTS C++ as in the following example.

```
__parent->wpModifyPopupMenu(hwndMenu,hwndCnr,iPosition);
```

### Write and run a Makefile

```
.SUFFIXES:
.SUFFIXES: .cpp .obj .dll .idl .rc .res .pdl .def
.cpp.obj:
      icc /c /Gd- /Se /Re /ss /Ms /Gm+ /Ge- /Ti+ -I. -c $<
.idl.def:
     sc -p -sdef -S1000000 $(@B).idl
.idl.hh:
     sc -p -shh -mnoqualifytypes -S100000 $(@B).idl
all: ex1.dll
```

```
ex1.obj: $$(@B).cpp $$(@B).hh
ex1.dll: $$(@B).def ex1.obj ex1.res
        icc /B" /de" /Feex1.dll /Fmex1.map ex1.def ex1.obj
        rc -p -x $*.res $*.dll
        mapsym ex1.map
ex1.res: $$(@B).rc $$(@B).hh
        rc -r $*.rc $*.res
```

Running nmake on this creates an ex1.dll file.

### Test the Example

The following REXX cmd file creates an instance of EX1 on the desktop. When an EX1 instance is created, its popup menu includes the new EX1 option, and selecting it plays the DosBeeps programed above.

```
/* */
call RxFuncAdd `SysLoadFuncs', `RexxUtil', `SysLoadFuncs'
call SysLoadFuncs
say SysCreateObject("EX1","Ex1 Example","<WP_DESKTOP>","OBJECTID=<AnEx1>")
```

## Summary

Issues important to using DTS C++ to implement WPS classes were discussed and a simple example was given. The example was programmed using the IBM Visual Age C++ Compiler version 3.0 with CSD 3 installed and using a current development build of the IBM SOMobjects Toolkit. A dtswps.zip file containing the example shown here (with the fixed WPS IDL files and all necessary SOMobjects emitter support -- aside from what is provided by installing the VAC++ Toolkit support) is available internally within IBM on the Web page at /afs/austin/u3/shd/outbox/WWW/index.html. When the OS/2 Magazine article is published, dtswps.zip will be available from a Compuserve forum.

### Author Biogragraphy

Scott Danforth received his Ph.D. in computer science at UNC Chapel Hill in 1984. He joined the IBM SOM team in 1991, where he designed and implemented SOM's native mode C++ bindings, SOM's multiple inheritance model, SOM's derived metaclass support, SOM's Metaclass Cooperation Framework, and most recently, SOM's new parent method call model. He represented IBM in the working group that produced the CORBA C++ mapping specification, and worked with MetaWare, Visual Age C++, and others at IBM to design DTS C++ and and implement the SOM extensions that support it. His publications include the book "Objects for OS/2" as well as numerous journal and conference articles.