SOM Overview and the WPS

Presented by: Dan Kehn, IBM Software Solutions

CompuServe ID 74140,3263 (GO OS2DF1)

Based in part on *Creating Objects for the Workplace Shell* by Joe Coulombe

Legal Notice

References:

GBOF-2254 *OS/2* Version 2 Technical Compendium* ("Redbooks")

The complete set of five volumes. All five volumes can be ordered separately. The two that are most useful for SOM and Workplace Shell programming are:

GG24-3732 *OS/2 Version 2.0 - Volume 3: Presentation Manager & Workplace Shell*
GG24-3774 *OS/2 Version 2.0 - Volume 4: Application Development*

S10G-6265 *OS/2 2.0 Presentation Manager Programming Reference, Volume II*

Contains all the WinXXX() calls, the Workplace Shell classes, and their wpXXX messages. This is also available on-line (PM Reference) in the *Information* folder that comes with the OS/2 Toolkit.

S10G-6309 *OS/2 2.0 System Object Model Reference*

This is also available on-line (SOM Reference) in the *Information* folder that comes with the OS/2 Toolkit.

G362-0001-14 *OS/2 Developer Magazine* - No. 3 1992

Article: *The OS/2 Workplace Programming Interface* by Mary A. Wright
(This is the issue with the OS/2 medal on the front, and the first issue following the name change from Personal Systems Developer.)

Ten Easy Steps to WPS Programming

1. **Design the objects**
2. **Design the object's views**
3. **Decide the SOM/WPS class to subclass**
4. **Create a class definition for the new object**
5. **Compile the class definition**
6. **Add the object specific code**
7. **Compile and link**
8. **Register the object with the Workplace**
9. **Create an instance of the object**
10. **Test and iterate**

References (cont'd):

G325-0650 *Client/Server Programming with OS/2 2.0, 2nd Edition*
by Robert Orfali and Dan Harkey.

ISBN 0-442-01522-4 *OS/2 2.x Notebook, the best of OS/2 Developer Magazine*
Article: *Object-Oriented Programming*
by Roger Sessions and Nurcan Coskun

G362-0001-17 *OS/2 Developer Magazine* - Vol. 5 No. 2 Spring 1993
Article: *Workplace Shell Programming Using Multiple Processes*
by Richard Redpath, Joe Coulombe, and Sue Henshaw
(This is the issue with 8 "earths" falling into a spinning vortex on the cover.)

ISBN 0-672-30240-3 *OS/2 2.1 Unleashed*
Chapter 5, "Workplace Shell Objects"

Source Code *IS* Provided

- **README.TXT**
- **WPS-PGM.TXT**
- **Lots of comments in code**
- **SHARE93 package on OS2TOOLS**
  - **SHR93.ZIP on CompuServe in OS2DF1, section 3**

The README.TXT file included with the example has an overview of all the files and other hints and tips. Much of it was based on comments that are in the source code.

Also see the WPS-PGM.TXT file which documents common WPS programming problems and their solutions.

But First, Some Assumptions About You...

- **Have used OS/2 2.x and the Workplace**
- **Understand basic object-oriented concepts**
  - **Data encapsulation**
  - **Inheritance**
  - **Polymorphism**
- **Some Presentation Manager programming experience**

Several assumptions about your prior experience: (1) You have used OS/2* 2.x and the Workplace Shell, and (2) you understand basic object-oriented (OO) programming concepts, and (3) you have some experience programming to the OS/2 Presentation Manager.

Some object-oriented programming concepts:

Data encapsulation is the hiding of the data's internal representation from programmers (clients) of the data. This is accomplished by providing APIs that query and manipulate the data without exposing its underlying structure. It is good programming practice since it shields the programmer from changes in the underlying implementation.

Inheritance is an OO programming mechanism for reusing and modifying the behavior of existing code.

Polymorphism is an OO programming mechanism which allows client code to treat common objects in a similar manner without concern for the underlying implementation.

Object-oriented programming use these principles to more closely model the real world and to help ease the burden of coding changes.

SOM - A Quick Explanation

- **"O" is Object**
  - **Data and behavior**
- **"OM" is Object Model**
  - **Representation of an object**
- **So, "SOM" is System Object Model**
  - **Common representation of an object model**

An object is data and behavior. Objects communicate using a messaging technique that varies among language implementations. These messages can be used to query an object's state or ask it to perform some behavior.

An object model is a representation of an object. In rough terms, the object model of C++ is a structure with an array of associated functions that act on that structure. In Smalltalk, the object model is either byte data, or pointers to other objects with a collection of methods that act on the data.

So, *System Object Model* is a common representation of an object model.

The goal is to allow for communication between objects and sharing of object class implementations independent of the underlying implementation language.

SOM provides a basic class definition compiler that generates bindings that are acceptable to the various compilers (C Set/2 and C++). SOM also provides a base class hierarchy which includes object management.

The WPS uses SOM as the basis of its object model.

A Quick Analogy of OOP for PM Programmers

- **WinSendMsg $\rightarrow$ somResolve**
- **WinSubclassWindow $\rightarrow$ parent: in CSC file**
- **WinCreateWindow $\rightarrow$ somNew**

PM is crudely object-oriented in that it provides some data encapsulation (window structures hidden by APIs), messaging (WinSendMsg), and crude inheritance (WinSubclassWindow). Some find it easier to understand truly OO implementations like SOM in terms of PM, but be forewarned of the shortcomings of this analogy.

Each case statement in a window procedure is like a method, ie, it handles a specific request for information or a request to take action. PM uses *WinSendMsg* to send messages; WPS/SOM uses *somResolve* to invoke methods.

The SOM compiler generates macros that mask out some of the less interesting parameters passed to somResolve, but you can see their ugly entirety in the .H file produced by SOM compiler when it processes the class's CSC file (see \TOOLKT20\H\WPOBJECT.H for an example).

PM provides several techniques for finding a given window. You can use WinBeginEnunWindows/ WinGetNextWindow/WinEndEnumWindows, WinQueryWindow (get parent, sibling), and root windows like HWND_DESKTOP and HWND_OBJECT.

WPS also provide techniques for finding a given object. WinQueryObject accepts object IDs (eg, "<WP_DESKTOP>"). WPObject class methods like wpclsQueryObject, wpclsQueryFolder, and wpclsFindObjectFirst/ wpclsFindObjectNext/wpclsFindObjectEnd can be used to get a pointer to a specific object, similar to how WinBeginEnumWindows(HWND_DESKTOP) can be used to get a specific window handle. SOM class information methods like somFindClass are similar to PM’s WinQueryClassInfo.

WinSendMsg can send a message across process boundaries, ie, it handles the context switch and serialization. Regrettably, somResolve does not. So wp methods (which are invoked via SOM's somResolve function) can only be invoked by those who are already running under the same process as the object was created, in this case the WPS process. Note, however, that new versions of SOM announced in the SOMObject toolkit solve this problem; WPS still uses the single-process version of SOM.

The WPS provide a few functions that handle the process switch for you and allow limited interaction with WPS objects from any process, eg: WinQueryObject, WinSetObjectData, WinDestroyObject, and WinCreateObject (notice there is no "WinQueryObjectData", which would be very helpful).

When you register a WPS class with WinRegisterObjectClass, WPS loads your class DLL. When an instance of your class is created, it binds your class to the SOM runtime in the WPS process, hence instances of your WPS-derived class always run under the WPS process, so all the wp and wpcls methods are available to your class and its instances. In a sense, it is kind of like writing a public window class, ie, PM loads your window class DLL on WinCreateMsgQueue; WPS loads your class DLL on the first invocation of wpclsNew specifying your class.

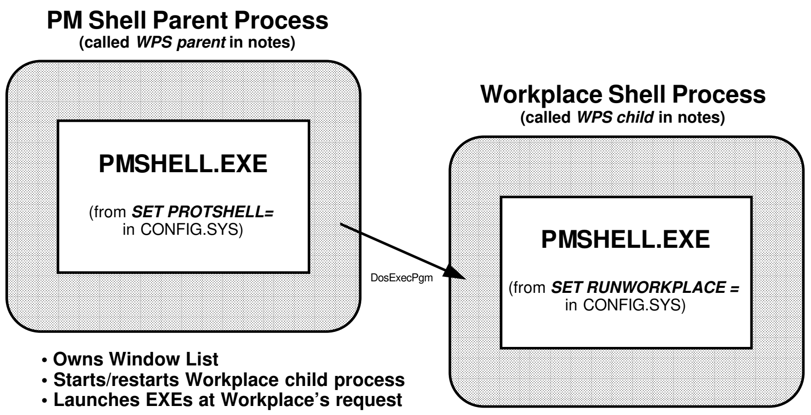
SOM in OS/2 2.x

- **SOM provides many binding options**
  - **offset resolution (much like C++ virtual tables)**
  - **name lookup (much like Smalltalk)**
- **SOM is dynamic-link library based**
  - **Methods of each class are in a DLL**
  - **Class/DLL is registered with SOM, when needed it is loaded**
- **SOM version used by OS/2 2.x is single-process**
  - **Later versions of SOM support distributed objects**

SOM allows the developer to choose the type of message resolution. The default is similar to C++ virtual tables, but SOM can also use a name lookup approach much like Smalltalk.

SOM in OS/2 is DLL based. The developer registers a class name and DLL name with the SOMClassMgrObject so instances of the class can be created. WPS subclasses the SOMClassMgrObject to add dynamic loading (when the first instance is created) and unloading (when the last instance is freed) of a class.

Workplace Shell Overview - Two Process Model


PM Shell Parent Process
(called *WPS parent* in notes)
PMSHELL.EXE
(from *SET PROTSHELL=* in CONFIG.SYS)
DosExecPgm
Workplace Shell Process
(called *WPS child* in notes)
PMSHELL.EXE
(from *SET RUNWORKPLACE =* in CONFIG.SYS)

- **Owns Window List**
- **Starts/restarts Workplace child process**
- **Launches EXEs at Workplace's request**

- **Creates SOM objects based on filesystem and OS2.INI**
- **Asks parent process to launch EXEs**

The Workplace Shell executable, PMSHELL.EXE, is used for two purposes. First, to be a rudimentary PM shell that handles the Window List. Secondly, to be the Workplace Shell itself. PMSHELL.EXE checks an environment variable (WORKPLACE__PROCESS=YES|NO) to determine which role it is to fulfill.

Note: OS/2 defines a parent/child relationship between processes. When a process starts another process, it is made the parent of that process. If the parent process terminates, the child process is terminated. If the child process terminates, whether abnormally or normally, the parent process is notified.

Workplace shell uses this two-process parent/child model to make the shell more reliable. If the WPS child traps, it can be restarted by the parent. The WPS child also asks its parent process to start new programs (e.g., when the user double-clicks the *Enhanced Editor* icon in the *Productivity* folder) so the newly-started program will not be terminated should the WPS child trap.

Why all this concern? Well, the WPS child uses the single-process version of SOM, resulting in potentially hundreds of objects being created who's methods are implemented by many different classes. The code for each class is in a dynamic-link library written by IBM* or any other WPS-enabled vendor. OS/2 APIs and the WPS use the exception handling capabilities of OS/2 2.x, however one badly coded class can bring down the whole WPS child process (similar to the troubles an error in a public PM window class can cause).

By keeping the WPS parent simple, there is little risk it will trap. It only loads IBM-supplied DLLs, so there is less chance of corruption. The WPS parent is notified if the WPS child abnormally terminates and can restart it.

Workplace Shell Overview - Opening a Folder

- **Query file system, create instance of corresponding class**
  - **.CLASSINFO extended attribute for data files in file system**
  - **OS2.INI's *PM_Abstract:Objects* entry for non-file system**
- **Object's record is inserted into PM container control (FID_CLIENT)**
  - **Most of WPS is PM dialog and window code**
  - **Most of remainder is object/class management and object persistent data management**

The WPS provides a visual hierarchy of folders that map directly to the file system (directories). Each folder (directory) contains data files that can be of different WPS/SOM registered classes. In addition, classes whose persistent state are not stored in a data file can be registered with the WPS (however, the APIs for doing so are private today). Such "base" classes are responsible for remembering which folder each of their instances belong.

For example, "abstract" objects like the *Color Palette*, *System*, and *Program* are stored in the OS2.INI file. Their instance data is in the OS2.INI application *PM_Abstracts:Objects*; their folder location is stored in OS2.INI application *PM_Abstract:FldrContent*.

So-called "transient" objects like those in the *Minimized Window Viewer* and printer object's queues are stored in memory. The transient class is responsible for keeping track of which folder each of its instances reside in so it can respond if later asked during folder population.

An obvious and reasonable consequence of this implementation is that a WPS object can only exist in one folder. However, it may be displayed in several views of the same folder.

Step 1: Design the Object(s)

- **Design each object as a class**
- **Define what each object will do**

Since objects are simply data coupled with behavior, it makes sense to start by answering the following questions: (1) What data is needed? (2) How should it be modified?, (3) What behaviors are needed?

An object that has no visual component is often called a *model*. Model code is the “business logic” of the data. Since the data of a business rarely changes (e.g., there will always be bills, invoices, ledgers) and the rules that change that data rarely change (e.g., rules for allowing a debit and credits don’t change), model code tends to remain the same over time.

Views, however, which present the data to the end user, change very often. Views change with each advancement of computer technology (witness the changes from character based user interfaces, to GUIs, then multimedia) and drives user demands to see data in a different way (tabular, chart, voice, video).

With this in mind, spend time in the beginning verifying your models accurately represent your business. You will enjoy the fruits of your labor for a long time!

Views should be designed with flexibility in mind because they change frequently. Encapsulating view code in an object class can help.

Note: The provided example view code is *not* encapsulated into an object class. This is left as an exercise for the reader.

Step 1a: Design Each Object as a Class

- **Person**
- **Address book**

Two model classes are defined in this example, Person (ShrPerson) and Address Book (ShrAddressBook).

Step 1b: Define What Each Object Will Do

- **Person (ShrPerson)**
  - **Holds pertinent data about a person**
  - **View should be succinct**
- **Address Book (ShrAddressBook)**
  - **Contains persons**
  - **Several possible concurrent views**
  - **Should be quick to find a given person**

ShrPerson represents the pertinent data about a person (name, address, city, state, zip code, and phone). Its view should be succinct since the user will want to find information about the person quickly, e.g., their phone number.

ShrAddressBook will contain persons. It will display them in several possible views. These views should allow the user to find a given person quickly.

Both of these models are simple. The address book model is simply a holder of persons.

Step 1c: Focus on the User’s Conceptual Model

- **Person is an easily recognized metaphor**
- **Address book is an easily recognized metaphor**
  - **Others may require more thought and testing**
  - **Consult domain experts**
  - **Consult less experienced users**

ShrAddressBook’s views should appear familiar to the user.

Models for user interface objects are not always easy to identify. Find representations that make sense from the user’s point of view, and avoid computer artifacts (files, directories, drives). Accountants work with ledgers, attorneys work with contracts, sales managers work with register sales, car salesmen work with sales worksheets -- all these examples can be modeled in the user interface to take advantage of the user’s prior experience.

Many developers have a tendency to design the user interface (view) first without regard to the user model. The user model (data and domain) should be designed since it changes the least. A well designed user model will result in a better designed user interface.

Domain experts can offer unique insight when designing the UI models for your business. However, less experienced users in the field domain should be consulted to verify the model makes sense for them, too, since training costs are a significant portion of a business’s yearly expenses.

Step 1d: Define Interactions with Other Objects

- **Create direct-manipulation tables**
  - **All objects you have defined**
  - **Include representative objects in the system**
- **Four interaction tables**
  - **Default (no keyboard augmentation)**
  - **Copy (Ctrl+mouse button 2)**
  - **Move (Shift+mouse button 2)**
  - **Link (Ctrl+Shift+mouse button 2)**
- **Sometimes a fifth interaction table**
  - **Create** (usually Create table is the same as Copy table)

Each object should be evaluated for its behavior as the source of a drag, and as the target of a drag.

Start by creating direct-manipulation tables which includes all of the objects you've defined plus some representative objects in the system, eg, Folder, Printer, and Shredder.

The purpose of direct-manipulation is to transfer data. Your tables should specify what data is transferred for each interaction. Create a separate table for each of the four interactions referenced above.

Step 2: Design the Object's View

- **Determine the views needed of each object**
- **Consider the information to be presented in each view**
- **Choose appropriate controls for each piece of information**
- **Design an effective and aestetically pleasing layout**
- **Design the menus (and pop-up menus)**
- **Iterate and test**

It is tempting to design the object's view first. But often when designing views first, the rest of the design is modified to serve the view's needs. However, the view (look) of the object frequently changes, resulting in unnecessary changes in the model.

Consider the data that the user needs first, then how objects that represent that data will interact. Additionally, consider how the user thinks of the data they use day-to-day, and what would be reasonable interactions among the different data (e.g., dragging a person object to a room object makes sense, but not visa-versa).

Only then design the views for the objects. This separation of view and model will result in model code that rarely changes.

Step 2: Design the Object's View (cont'd)

- **If object's primary property is containment**
  - **Decide limit of its contained objects, if any**
  - **Be certain limits in containment are reasonable to user**
- **Address book's default view is *Indexed View***
- **Person's default view is *Settings View***

Many objects are a composition of other objects, not a container of objects. For example, an *event* object could be composed of two objects, a *time* and *place*. But the event object is not a container since its primary purpose is not containment.

Folders are clearly a container object, since their primary purpose is to contain other objects. The address book's primary purpose is to contain person objects. However, the address book is not a general container, that is, it is an example of restrictive containment. It only allows objects of a specific type within it.

There are other examples of restrictive containment objects in OS/2. The *Printer* object only allows print jobs within it; the *Minimized Window Viewer* only allows minimized windows within it.

Restrictive containment can effectively be used to guide the user to purpose of a container. But be certain that such restriction is considered reasonable to the user.

Step 3: Decide the SOM/WPS Class to Subclass

- **Three base classes provided by the WPS**
- **Findable objects are the basis for the hierarchy**
- **Model and view are not clearly separated in WPS hierarchy**
  - **Example code shows how to improve WPS model/view separation**

The root class for non-UI objects is SOMObject. The WPS defines base classes for UI objects. These base classes are all subclasses WPObject, which is a subclass of SOMObject.

UI objects that you define for the WPS should be a subclass of one of the three base classes so your object can be located by the system “Find...” and to inherit useful methods. Do not subclass from WPObject directly.

Hierarchy based on table 5.1, *OS/2 2.1 Unleashed*, with corrections:

- ***SOMObject***
  - WPObject
    - WPFileSystem
      - ***WPDataFile***
        - WPBitmap
        - WPIcon
        - WPPointer
        - WPProgramFile
          - WPCommandFile
        - WPMet
        - WPPif
      - ***WPFolder***
        - WPDesktop
        - WPStartup
        - WPDrives
        - WPMinWinViewer
        - WPFindFolder
        - WPNetgrp
        - WPServer
        - WPSharedDir
        - WPTemplates
        - WPRootFolder
    - ***WPAbstract***
      - WPClock
      - WPCountry
      - WPDisk
      - WPKeyboard
      - WPMouse
      - WPPalette
        - WPSchemePalette
        - WPColorPalette
        - WPFontPalette
      - WPShadow
      - WPShredder
      - WPSound
      - WPSpecialNeeds
      - WPSpool
      - WPSystem
      - ***WPProgram***
      - WPPrinter
        - WPRPrinter
    - ***WPTransient***
      - WPCnrView
      - WPFilter
        - WPFinder
      - WPMinWindow
      - WPJob
      - WPPort
      - WPPrinterDriver
      - WPQueueDriver

• The most commonly subclassed classes are shown in ***bold italics***. Some unnotable classes are omitted.

Step 3: Decide the SOM/WPS Subclass (cont'd)

- **Address book's primary purpose is containment**
- **WPFolder is a good choice for superclass**

WPFolder provides basic containment behavior. In addition, it provides three standard containment views: Icon, Details, and Tree.

Implementing alternative containment views like the address book's *Indexed View* requires WPFolder methods that were omitted from the OS/2 2.0 product. Therefore the example adds a WPFolder subclass, ShrFolder, which adds the shrAddedToContent and shrRemovedFromContent methods (the source code to these methods are not included in the package). Boca is aware of the problem and plans to address them after OS/2 2.1 ships.

Step 3: Decide the SOM/WPS Subclass (cont'd)

- **Person's best superclass is not as clear**
- **WPAbstract**
  - **Efficiently stores object who's data is small**
  - **Cannot be copied to diskette**
- **WPDataFile**
  - **Can be copied to diskette**
  - **Wastes space for small data (2-4K min. allocation)**
- **WPTransient**
  - **Only if objects can be created from another source**

ShrPerson subclasses from WPAbstract mainly because ShrPerson's data requirements are quite small. However, if ShrPerson typically used 2K+ data, it should subclass WPDataFile.

The wpSaveString, wpSaveLong, and wpSaveData methods used in WPDataFile's wpSaveState method store the data in the extended attributes of the data file. Other attributes are stored in the extended attributes, for example ".ICON" attribute holds the icon (if one has been explicitly set), ".LONGNAME" holds the name of data files over 8 characters on FAT file systems (HPFS supports long file names).

Subclasses of WPDataFile are free to store data in the data file itself. The wpQueryRealName method retrieves the full file name which can be used by DosOpen, DosRead, etc. The DosRead/DosWrite APIs are much faster than the DosQueryFileInfo call to retrieve the extended attributes. Extended attributes are currently limited to 64K bytes total (.ICON, .CLASSINFO), so store large data in the file itself.

Step 3: Decide the SOM/WPS Subclass (cont'd)

- **Put model code in a separate process**
  - **Greater stability, easier to develop large projects**
  - **Start with DosExecPgm during DLL initialization**
  - **Use inter-process communication**
- **View code can be in WPS process**

All WPS subclasses run in a single process. When an instance of a WPS subclass is created, the corresponding class dynamic link library (DLL) is loaded by the WPS.

It is recommended that WPS developers limit the size of the class DLLs loaded by the WPS by moving as much code as possible into a separate executable. This executable should contain the bulk of the model code. The class DLL should start the model executable in its DLL initialization routine and signal it to terminate in the DLL termination routine (see *IBM C Set/2 User's Guide*, page 234 for an example). The model object which exists in the WPS process can communicate with the model code running in another process using standard inter-process communication techniques (queues, semaphores, shared memory, PM object windows).

Subclasses of WPDataFile typically pass their file name to the model code in the other process for interpretation. Instances of WPAbstract can keep a key to pass to the model code (e.g., record number, database key).

Step 4: Create Class Definition for the New Objects

- **Define instance data needed**
- **Define methods to query/set instance data**
- **Define methods needed by others**

A model’s design is driven by its data. So define the data the model needs first, then the necessary access methods for this data.

Once the data is defined, consider what methods are needed by other objects (clients) . Lastly, consider what methods are needed to support the various views of the model.

You should review the methods provided by your superclass to determine if you can reuse any of them.

Step 4: Create Class Definitions (cont'd)

- **Define "helper" objects**
- **Helper objects are usually non-UI objects**
  - **Ordered list object (ShrList)**
  - **Notifier object (ShrNotifer)**

"Helper" objects are a great way to get code reuse! You will get more reuse by combining the functions of several objects (a *has-a* relationship, in OO parlance) instead of having a larger single object with many characteristics (a *is-a* relationship).

ShrAddressBook and ShrPerson both have a single instance of ShrNotifier. They implement one method, shrQueryNotifier, and let the notifier handle the other messages (shrAddInterestedWindow, shrRemoveInterestedWindow, et al).

Notifier (Step 4, cont'd)

- **Used to synchronize view with changes in model**
  - **shrAddInterestedWindow**
  - **shrRemoveInterestedWindow**
  - **shrNotifyInterestedWindows**
- **More flexible than that provided with WPS**
  - **VIEWITEM structure of object's use list**
- **Example of "has a" versus "is a"**
  - **Client only has to implement shrQueryNotifier method**
- **Encourages model and view separation**

One model can have several different views on it opened at a time. For example, the user can open an Icons, Details, and Tree on a single folder and they expect each of the views to remain synchronized (i.e., adding an object to the folder in one view should display it in the other views of the same folder).

WPS provides a mechanism for tracking what views are open on an object. Each WPS object has a *use list* of all the resource allocated to the object (e.g., memory, container records). Included in the use list are *view items* that have view type and PM frame handle to each view opened on the object. The view items can be used by the model to notify the views when a change in the model data occurs (which was likely caused by one of the open views, e.g., the user adds an object to an open view of a folder).

ShrNotifier maintains a list similar to the view items of an object's use list. but is different in that it can be used to notify an arbitrary window, not just a frame window. This is used in the ShrPerson to keep the *Info* page of the Settings notebook synchronized with changes in the object title (see ShrPerson's wpSetTitle method in PERSON.C, and the SHRN_PERSONNAMECHANGED case in ShrPersonInfoPageDlgProc).

Step 5: Compile the Class Definition

- **SOM compiler**
- **Produces binding files bridging SOM and development language**
- **Generates C macros and function skeletons**

The class definition file defines the superclass, instance data, and methods. Methods are inherited from the superclass and can be overridden by specifying the override keyword, e.g.:

```
  override wpModifyPopupMenu;
```

The above example shows how you would override the wpModifyPopupMenu method which is defined by WPObject. When overriding methods, be sure to call the superclass (parent) method first, if you are adding behavior, not replacing behavior.

Step 6: Add the Object Specific Code

- **ShrPerson new methods**
  - **set/query, e.g., shrSetAddress, shrQueryAddress**
  - **shrQueryNotifier**
- **ShrPerson overridden methods**
  - **wpUnInitData**
  - **wpSaveState**
  - **wpRestoreState**
  - **wpAddSettingsPages**
- **ShrPersons's view code**
  - **handles *Info* page dialog created by wpAddSettingsPages**

The example code is structured to separate model and view. The PERSON.C file has the model code and PERSONV.C has the view code. There are some model methods that affect the view, e.g.:

- wpOpen: opens the view
- wpAddSettingsPages: add pages to the Settings view notebook.
- wpModifyPopupMenu: add items to the pop-up menu
- wpFilterPopupMenu: add/remove standard pop-up menu items
- wpclsQueryIconData: query default icon for class
- wpclsQueryDefaultView: query default view for class.

Separate code files for model and view code clarifies the responsibilities of each and decreases the tendency to mix the two.

Step 6: Add the Object Specific Code (cont'd)

- **ShrAddressBook new methods**
  - **shrQueryNotifier**
- **ShrAddressBook overridden methods**
  - **shrAddedToContent**
  - **shrRemovedFromContent**
- **ShrAddressBook's view code**
  - **handles wpOpen**

ShrFolder, a subclass of WPFolder, adds notification methods for when an object is added/removed from the folder. These methods give the model an opportunity to synchronize its data with its views.

ShrAddressBook overrides these methods to detect when a person object is added or removed from the address book (via drag/drop, or selecting a choice like *Copy...* from the person's pop-up menu).

Step 7: Compile and Link

- **SOM compiler**
  - ***.C skeleton source code**
  - ***.SC - public compiled version of *.CSC methods**
  - ***.PSC - private compiled version of *.CSC methods**
  - ***.IH - implementation header**
  - ***.H - public header**
  - ***.PH - private header**
- **Standard C compiler, linker**
- **Create DLL's**
  - **One per class, or combined**

The SOM compiler takes a class definition file (*.CSC) and produces:

*.IH - implementation header to be included in the *.C file which implements the methods.
*.H - public methods header
*.PH - private methods header
.SC - compiled SOM class definition (.CSC)
*.PSC - private portion of compiled SOM class definition
*.C - skeletal source code

These files can be compiled with IBM C Set/2. Messages can be sent to a SOM object using macros generated by the SOM compiler (which call the SOM message resolution function *somResolve*).

Step 8 and 9: Register with WPS, Create Instance

- **WinRegisterObjectClass API**
  - **Registers with WPS, WPS then registers with SOM**
- **Loads the DLL**
- **Makes the new class available**
  - **Appear in *Templates* folder**
  - **See example Address Book installation**

An installation executable should be provided with your class DLLs. WinRegisterObjectClass returns only TRUE or FALSE, so the SHRINST.EXE , provided with the example, checks for the most common mistakes before calling WinRegisterObjectClass.

Once a class is registered with the WPS, it will appear in the *Templates* folder unless the class style includes CLSSTYLE_NEVERTEMPLATE. Note that the class style of WPAbstract includes this style; this can be overridden in your subclass's implementation of wpclsQueryStyle, if desired.

Step 10: Test and Iterate

- **Test each object for code quality and usability**
- **Make sure each object interacts gracefully with others on the Workplace**
- **Don’t be afraid to iterate several times**
- **Challenge your design assumptions**

Test each object in isolation, then together. Involve beta testers early to help find problems in code quality and usability.

Experience has shown that two or three beta test cycles are adequate for most designs.

Advantages of WPS Programming

- **Common look and feel**
- **Simplifies initiation of drag**
- **Simplifies pop-up menu creation and emphasis**
- **Good support of “perfect save” model**
- **Simplifies Settings notebook creation**
- **Provides object storage reclamation (awake/dormant)**
- **Provides persistent object handles**
- **WPFolder behavior quite useful**

The CUA’91* look and feel of the WPS is much easier to reach if you program directly to the WPS. And the WPS hierarchy provides significant help. For example, the WPS container owner subclass winproc handles many of the container notification messages; it sets up drags on CN_INITDRAG, including “stacking” of the drag images; it helps create and display the pop-up menu of one or more selected objects on CN_CONTEXTMENU.

The WPS also provides support for handling “perfect save”, that is, the automatic saving of object data to persistent storage and it does this on a separate thread (the so-called “lazy write” thread).

Creating settings notebook is simple - just provide a dialog procedure and a dialog resource. WPS handles sizing the notebook tabs and loading the dialog when the user chooses the page.

The WPS “snoozer” thread implements object storage reclamation. When an object is no longer locked nor displayed in a container for a few minutes, it is automatically made dormant (wpMakeDormant) to free the object’s storage.

Linking objects together across reboots is simple with persistent object handles provide by the WPS (wpclsQueryObject(_WPObject, hObject)).

The WPFolder makes displaying objects in a container fairly easy. Details view is easy to modify, once you understand how (see SHARE93 package on OS2TOOLS).