Software Extensibility and the System Object Model (SOM)

Preliminary

Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

CHAPTER 1

Software Extensibility and the System Object Model (SOM)

Contents

To make object-oriented libraries viable for extending software, Copland uses IBM's System Object Model (SOM), a new model for developing and packaging object-oriented software. This document introduces you to where and why Apple uses SOM classes in Copland and what impact, if any, these may have on your own software development.

The Apple implementation of the System Object Model is called SOMobjects for Mac OS. In short, this technology provides Copland with an object-oriented mechanism for software extensibility without the drawbacks commonly associated with object-oriented programming—in particular, the inability to reuse binary code, and various language incompatibilities between class libraries and the applications that use them.

In Copland, SOMobjects for Mac OS is used to implement

- interface definition objects (IDOs) for many standard user interface elements such as windows and menus
- panels for other human interface elements such as controls, lists, and icons, which incorporate such standard behaviors as keyboard navigation, copy and paste, and drag and drop
- all Text Service Manager services, including interactive text services (like spelling checkers) and text input methods (like user keyboard activity)
- runtime support for OpenDoc component software

By using SOMobjects for Mac OS, Copland provides users with up-to-date features and a consistent user interface across applications, system software releases, and application revisions. SOMobjects for Mac OS benefits you and your users in these main areas:

- When Apple adds new features to subsequent versions of the Mac OS, users won't need to update their software to gain these features, because applications will automatically inherit them. For example, if Apple defines a new capability for windows in a future version of the Mac OS, all Copland applications would automatically gain this new capability. With SOMobjects for Mac OS, Apple provides an easier way for you to keep your products up to date.
- When applications enhance or modify portions of Copland, these enhancements will be available to users even after Apple releases later versions of the Mac OS. For example, an application can alter the default behavior of a control provided by Copland in one release, and this modified behavior will continue to work in subsequent releases of the Mac OS. With

SOMobjects for Mac OS, Apple provides an easier way for you to extend portions of Copland while remaining compatible with future versions of the Mac OS.

- SOMobjects for Mac OS provides you with an object-oriented approach to extensibility in your own code. For example, you can create your own SOM classes that can be easily altered and enhanced in subsequent updates to your application.

The Code Fragment Manager forms the foundation of the Copland runtime environment. The Copland implementation of SOM is layered on top of the Code Fragment Manager. The SOM kernel—that is, the runtime portion of SOMobjects for Mac OS—is an application-level shared library. Because the SOM kernel is implemented as a standard application-level shared library, each program using SOMobjects for Mac OS is completely independent from all others. (The SOM kernel consists of the classes SOMObject, SOMClass, and SOMClassMgr, their methods, and a number of class-independent functions and macros. In Copland, the SOM kernel resides in two files: the runtime shared library and the shared library referenced at link time.)

The Need for Software Extensibility

Extensible software is designed to be more easily expanded, modified, and updated—either by its creator or by other programmers. Because the code that creates and manages controls is extensible in Copland, for example, you can easily tailor controls in a manner appropriate to your application to make them more helpful to users, and Apple can easily modify or add new control capabilities in later Mac OS releases.

Extensibility was given little consideration in the designs of earlier versions of the Mac OS, but developers found ways to extend system software on their own—with useful but sometimes unfortunate results for users. For example, while all previous versions of the Macintosh Toolbox have provided useful programming interfaces to help programs manage such human interface elements as menus, many application developers needed features not provided by the Macintosh Toolbox, so they wrote their own code to create such extensions as pop-up menus and tear-off menus. Unfortunately, this sort of ad hoc system software reengineering has caused a multitude of problems, including

- inconsistency to users—as, for example, when they must learn the different appearances and behaviors of all the custom implementations of tear-off menus provided by many different applications
- unnecessary development work for you—as, for example, when you needed to create your own version of tear-off menus instead of relying on code that the Macintosh Toolbox could have provided
- additional testing and qualification burdens for you—as, for example, when you needed to ensure that your own version of tear-off menus worked correctly under a multitude of software and hardware configurations
- system and application instability—as, for example, when custom tear-off menus created by other programmers (not by you, of course) did *not* work correctly under certain software or hardware configurations, causing programs to crash
- revision constraints for the Mac OS—as, for example, when you or others created user interface elements that have made it difficult for Apple to implement or update them in a uniform way across all applications

To help alleviate these problems, Copland implements many capabilities in more easily extensible SOM classes. Note that SOMobjects for Mac OS is not the only mechanism Copland provides for extending or updating software. Other Copland mechanisms are described at the end of this document. However, SOMObjects for Mac OS does provide an ideal mechanism for designing and packaging extensible software using object-oriented programming techniques.

The Object Oriented Approach to Extensibility

As developers using object-oriented programming techniques know, object classes facilitate the addition of features and capabilities to existing source code. For example, without changing any other code in a drawing program, you can override methods in the class for an object that draws itself as a two-dimensional black-and-white square so that the object can instead draw itself as a three-dimensional color cube.

However, commercial object-oriented languages such as C++ suffer because they don't support the reuse of binary code—they support the reuse of source code only. For example, to make use of the object that can draw itself as a three-dimensional cube, you would probably need to recompile the entire application. To update users with this new three-dimensional drawing feature,

it would be much simpler if you could simply distribute an updated class library instead of sending a completely recompiled application.

The Benefits of the System Object Model

The System Object Model (SOM) is most useful for providing an object-oriented programming interface to a shared library. This model supports data encapsulation, inheritance, and polymorphism—the key characteristics of object-oriented programming.

However, unlike other class types (such as C++ classes), SOM classes provide release-to-release binary compatibility. In the future, for example, Apple might add new capabilities to the SOM class for the standard window IDOs. Apple can easily update Copland windows by replacing the binary library for a window IDO, and—without being recompiled or relinked—applications will automatically inherit the new window appearance and behaviors. Without the System Object Model, it is difficult for programmers using one object-oriented language to produce shared libraries for use by other object-oriented languages while also maintaining binary compatibility from one release of a product to the next.

While compilers for object-oriented languages produce class libraries that are incompatible with different languages, the SOM approach to object-oriented programming provides compiler and language independence. Binary class libraries can be created in multiple languages—including procedural languages like C as well as object-oriented languages like C++. These libraries, in turn, can be used—and even subclassed—in different languages. For example, an Apple engineer can use her favorite language to write a SOM class for Copland, and you can use your favorite language to subclass and modify this class. Better yet, an application written in another language can link with the library for this newly modified class.

As you can begin to see, the System Object Model is not a complete implementation language or programming system. Instead, it complements existing languages with which you are already familiar and productive.

Using the SOM Classes Provided by Copland

The SOM classes provided by Copland allow Apple to update elements of the Mac OS without forcing users to reinstall all of system software. Applications using those elements automatically incorporate the new features without needing to be recompiled themselves. For example, if Apple updates the look of windows in the future, all Copland applications will automatically inherit the updated look.

You can also use the SOM classes provided by Copland to extend Copland features. For example, if you find a compelling reason to create a new type of control, you could subclass the panel for a standard control and override its methods to provide the look and behavior you need for your application.

The majority of developers, however, will use the SOM classes provided by Copland without modifying them. To use the standard appearance and behavior of windows, for example, you can use your preferred language, which can be a procedural language like C and Pascal or an object-oriented language like C++, to call the programming interfaces provided by the Copland Window Manager. The Window Manager in turn uses a standard window IDO when called by your application. The IDO actually draws the windows.

Using Unmodified SOM Classes

To incorporate an unmodified SOM class library, such as one of the standard panels, you simply link the panel with the compiled version of your source code to create a binary executable file.

Most developers won't need to modify any of the SOM classes provided by Copland because they incorporate most of the features that developers have created for themselves in the past; for example, the Copland Toolbox provides such common (but previously nonstandard) interface features as floating windows, keyboard equivalents in menus, and tear-off menus. (Even applications that have already created these features in System 7.5 should replace them with the Copland versions so that application features all share a consistent appearance, even when users switch between themes.)

IMPORTANT

While the majority of developers won't need to modify SOM classes provided by Copland, a significant number of developers probably will, in order to customize or extend Copland features. SOM classes may be created or subclassed in any language for which a developer has a SOM compiler. Therefore, if you develop compilers, Apple hopes that you provide SOM compilers for the languages your products support. ▲

Modifying SOM Classes

Only a minority of developers will need to alter the default behavior or appearance of elements that are implemented as SOM classes in Copland. For example, to create an entirely new control—such as, say, a throw switch—you can subclass a controls panel and then override its drawing methods. You then link the subclassed panel with your application.

The programming interface to an object class is described in the Interface Definition Language (IDL), a language resembling C++. The IDL file for a class specifies the names of the methods that it supports, its return types, its parameter types, and other types of information. The IDL files for all SOM classes in Copland are available to you for development purposes.

To alter a class such as a standard panel, you can use an IDL compiler to generate an implementation template file containing function definitions for each method in the class. The IDL compiler provides emitters that output the implementation template file in various programming languages, such as C and C++. You modify the implementation template file to override any methods for the class. You then use a SOM compiler to create an object file, which you link to your application.

The Interface Definition Language provides a cross-language transportation mechanism. On other platforms, IDL compilers are quickly being supplanted by direct-to-SOM compilers that allow creation of SOM object files without the interim steps involving the IDL compiler and IDL files. Apple hopes that you will help make direct-to-SOM compilers available for Copland.

To create a subclassed control at runtime, the binary application file uses the programming interfaces defined by the Control Manager, which in turn uses the subclassed panel linked with the application.

Creating SOM Classes

SOMObjects for Mac OS provides you with an object-oriented approach to extensibility in your own code, too. You can package application features in SOM classes, allowing you to more easily alter and enhance these features in subsequent product revisions. For example, the developer of a tax-preparation application could implement tax calculation code in SOM classes for easier modification every year when the tax laws change.

Another benefit for you is that the System Object Model is an emerging industry standard being implemented on most major operating systems. This simplifies cross-platform development, because you can package application code in SOM libraries, which are then easily ported across systems. For example, the developer of a tax-preparation application could package the code that performs tax calculations as SOM class libraries in Copland. The developer would not need to revise these libraries when creating products that run on other SOM-supportive operating systems.

To create a SOM class, you can create an IDL file using the Interface Definition Language, use the IDL compiler to create an implementation template file, and use a SOM compiler to create an object file to link with your application, or—as direct-to-SOM compilers become available—you can use a direct-to-SOM compiler to directly create the object file.

Other Mechanisms for Extending Software in Copland

SOMObjects for Mac OS is not the only mechanism Copland provides for extending or updating software. Other ways that you can extend system software include

- using Collection Manager collection tags
- calling Patch Manager functions
- creating background-only programs

Other ways that you can design extensibility into your own products include

- using OpenDoc components within your application

- separating code into replaceable, dynamically linked libraries
- separating your products into application programs and background-only programs.

IMPORTANT

In System 7.5, developers often modify system software by creating files of type `'INIT'` and patching a system trap table to effect global changes for all applications. However, Copland does not support these System 7-type system extensions. Because Copland's runtime environment includes a virtual memory system, multiple address spaces, and concurrent processing (all of which make the system highly dynamic), it does not clone a systemwide global state when launching applications. Therefore, these System 7-type system extensions are unable to simultaneously modify system software for all applications. ▲

The rest of this document introduces the extensibility capabilities of shared libraries, collection tags, the Patch Manager, and background-only programs. See the latest developer release of OpenDoc for more information about creating OpenDoc components.

Note

The Apple Shared Library Manager (ASLM), a shared library technology available on some earlier versions of the Mac OS, is not supported in Copland. ◆

Shared Libraries

An application does not need to use SOMObjects for Mac OS to extend or update its code; it can instead add or replace code fragments packaged as dynamically linked libraries. All executable code and associated data in Copland is packaged in shared libraries. The Code Fragment Manager, in turn, loads these libraries into memory and prepares them for execution. (A dynamically linked library is a shared library that exports code or data that can be referenced by another fragment at execution time. All shared libraries, including dynamically linked libraries, are created at program link time.)

Separating software into different, dynamically linked libraries simplifies updating and extending the software. For example, if Apple wants to enhance

QuickDraw 3D in a later release of the Mac OS, Apple can simply replace the library for QuickDraw 3D instead of releasing a new version of the entire operating system. Similarly, you can divide your product into pieces that lend themselves to periodic updates and package these pieces as separate, dynamically linked libraries.

While dynamically linked libraries are useful for easily updating and extending software, the Code Fragment Manager does not inherently support object oriented interfaces. Built on top of the Code Fragment Manager, SOMObjects for Mac OS does provide an ideal mechanism for designing and packaging software when you want to use object-oriented programming techniques as well as provide extensibility in your code.

Collection Tags

Many developers who use standard Copland features may also need to add their own data to these features—as, for example, when an application uses a reference constant for identifying a window type. Because Copland hides the internal data structures for many of these features from programmers, Copland allows you to tag the features with data through the Collection Manager programming interfaces, which were first made available with the release of QuickDraw GX.

Collection tags are also an easy way for applications to change or enrich the attributes of such user interface elements as menus and windows. For example, an application can turn a regular menu into a tear-off menu by simply adding the tear-off tag to the menu with a single Collection Manager call.

Copland defines a standard set of collection tags—such as color, tear-off, keyboard equivalent, and title-bar icon tags. An application can modify the standard collection tags, and it can define custom collection tags—such as sounds to play, special action procedures, and pieces of data—for its own use.

The Patch Manager

In versions of Mac system software that preceded Copland, developers often modified the behavior of a particular system software manager by patching routines in its trap table. In Copland, however, such patching is heavily constrained.

First of all, PowerPC processors do not use trap tables. Therefore, when software patches A-traps, Copland must emulate the 68K processor and an A5 world. This appreciably slows software performance.

Secondly, because of the Copland runtime environment, it is very difficult to set global state data with a patch, thereby constraining it to work reliably only with local effect—that is, within only one process at time.

For these reasons, Apple discourages patching of system software routines. However, it is clear from experience that patching is sometimes necessary. Therefore Copland supplies a Patch Manager that defines a new patching API. When patching is necessary, this API simplifies your work and helps ensure that your patch is made correctly.

Using the Patch Manager, you can patch system software routines for one application at a time. A developer that packages a patch in a shared library and makes it available to all applications on the system allows the patch to be instantiated in every process that calls the library. However, even with the help of the Patch Manager, it is very difficult for a patch to set global state data.

The file "Patch Manager" (in the folder Developer Documentation: OS Documents) on this CD provides detailed information about Copland's new patching mechanism.

Background-Only Programs

You can also extend Copland by using server processes to create background-only programs. For example, you can create a background-only program that watches for electronic mail and alerts users to the arrival of mail. By supporting a server-client relationship with application programs, this background-only program could also help users read and reply to mail from within any application program. In this way, background-only programs can extend Copland with new or improved systemwide capabilities.

Similarly, you can enhance the extensibility of your products by separating them into application programs that interact with users and background-only programs that process data and perform time-consuming I/O and compute-intensive operations. Then, for example, if you wanted to extend your product's user interface capabilities, you could replace your application program and leave your compute-intensive programs untouched; likewise, you could update the computer-intensive portions of your product and leave your user interface portion alone.

See the file “Intro to Kernel and OS Services” in the OS Documents folder on this CD for more information about processes.