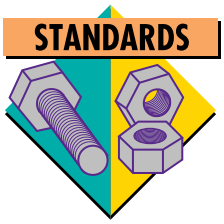


OpenDoc and Its Architecture

By Chris Nelson

OpenDoc and its related technologies represent an important standard for compound documents and component integration. IBM is both a member and a contributor of technology to the CI Labs consortium, holder of the OpenDoc technology. IBM is also producing the UNIX® reference implementation of OpenDoc. This article provides an overview of OpenDoc architecture and its related technologies: Open Scripting Architecture (OSA), Bento, ComponentGlue Technology, and System Object Model (SOM). It also discusses the programming model for developing software based on OpenDoc.

Compound document creation has undergone major changes during the last 25 years. With the advent of new technologies, a short history of compound documents can help put OpenDoc technology into perspective.

The 1970s—Documents by Hand

In the late 1970s and early 1980s, text, charting, and accounting tools were available to handle a wide variety of user needs. But each tool could work only with its own type of data and output that data in a limited fashion. If a document needed to include output from all of these tools, it was literally a cut-and-paste operation. If the document required any changes, it often meant starting from scratch, running each tool in succession, and hand-pasting the results together again (see Figure 1).

The 1980s—Tool-Level Integration

By the mid-1980s, platforms that enabled output from tools to be captured and placed in documents were readily available. One application or editor owned the entire document. Document layout was set aside for "pictures" of the embedded data, as shown in Figure 2.

Although this method was much easier than the manual cut-and-paste operations, this approach had several problems:

- The output data was static or "dead." If changes were required, the user had to return to the original tool, make the changes, then perform the electronic cut-and-paste operation.
- Generally, only pictures of the document could be traded across computers, since the

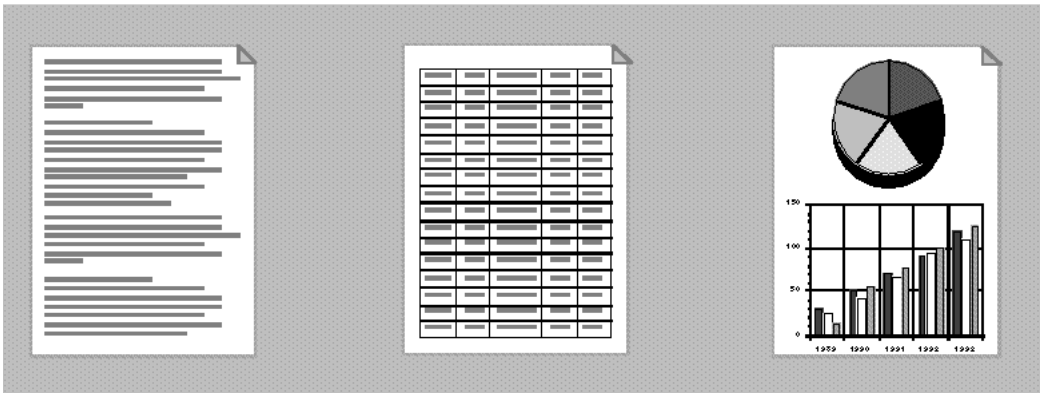
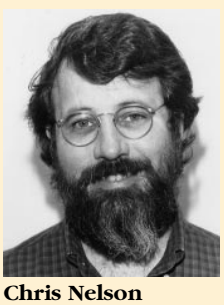


Figure 1. The 1970's—Documents were integrated by hand



Chris Nelson

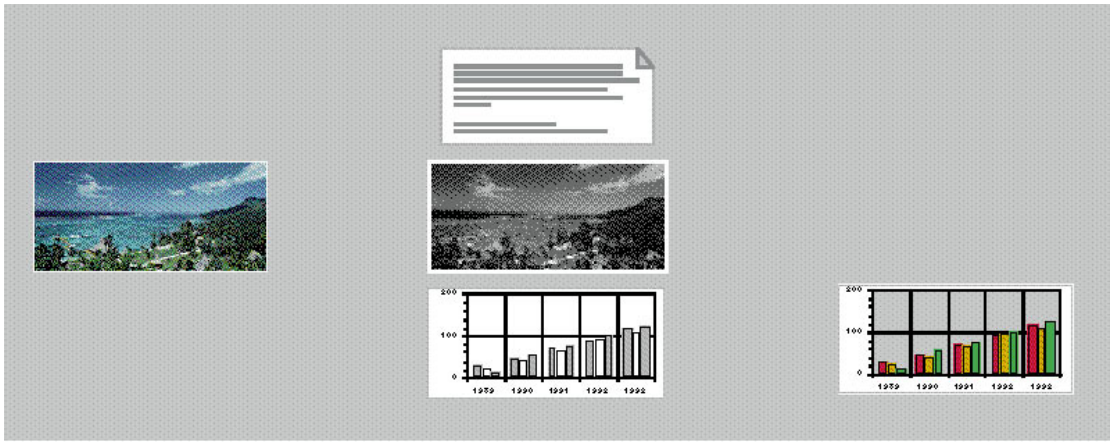


Figure 2. The 1980s—Tool level integration

document was tied to the local data and the tool.

- Items that could be placed into a document were limited to a small set of text and graphics, which were "pictures" of the data. There was no link back to the original data.

There was an additional problem with the tool-centric approach. As applications were able to support more and different kinds of data, the complexity of those applications increased significantly. Release cycles became stretched, requiring more programmers, increasing test time, limiting function between releases, and significantly increasing development costs. Applications also became complex and fragile.

The 1990s—Document-Centric Computing

With the advent of OpenDoc and other compound document architectures, such as Object Linking and Embedding (OLE) and the document framework in Taligent®, the model switched from tool-centric to document-centric.

In a document-centric computing model, users can focus on the documents they want to create rather than the output of a specific tool. Users can see a document that has several parts—all embedded and integrated into a single document, as shown in Figure 3. The type of parts that can now be included is dependent only on the availability of a part handler (editors and viewers). Some parts and features that we might first expect to see in these documents include the following:

- Formatted text
- 2-D and 3-D graphics
- Images

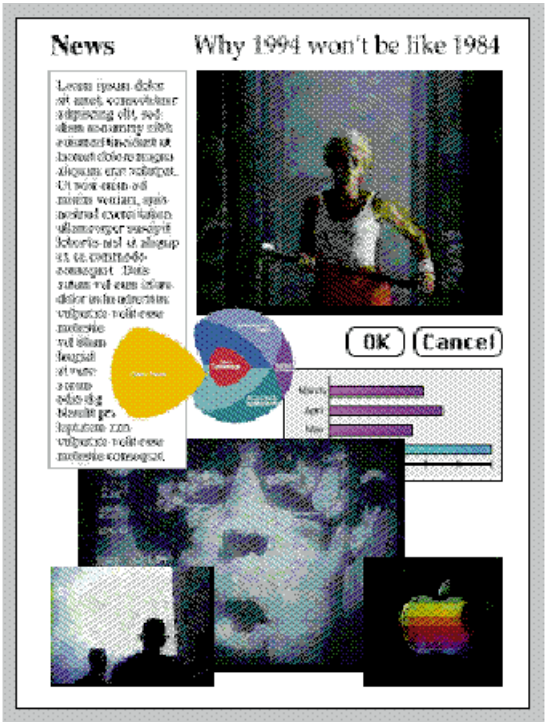


Figure 3. The 1990s—Document-centric computing

- Audio
- Video
- Structured data, such as appointments and calendars
- User interface control structures
- Inter- and intra-document links
- Document draft control

Part Handlers

The exchange of documents across platforms is greatly simplified, because the focus is on the various parts of the document—the data. The

only requirement is an available part handler (not a specific application) for the data contained in the document.

Applications are developed as small components called *part handlers*, which can be developed independently from other part handlers. In general, complexity is significantly decreased, allowing faster and less costly development cycles.

This approach also broadens the functionality that end users receive. Users can now acquire part handlers that integrate automatically, eliminating the dependence on one vendor for an integrated solution.

Compound Document Architectures

Currently, there are three available compound document architectures:

1. Microsoft's® Object Linking and Embedding (OLE)
2. CI Lab's OpenDoc and associated technology
3. Taligent's document framework

OpenDoc is a bridging strategy for application vendors who want to work in an OLE, OpenDoc, or Taligent environment with existing applications.

OpenDoc is a bridging strategy for application vendors who want to work in an OLE, OpenDoc, or Taligent environment with existing applications. With a quick port to OpenDoc, developers can extend the life of existing software products. They can also use OpenDoc to provide a migration path to Taligent-based products by introducing Taligent technology incrementally into their software base.

CI Labs

OpenDoc is an industry solution—not an IBM or an Apple® solution. The technology that makes up the integrated components of OpenDoc has been placed in an independent consortium: Component Integration Laboratories (CI Labs). CI Labs makes this technology, including the source code, available to all its members.

The initial technology base that will be contained in CI Labs consists of the five integrated OpenDoc components:

- **OpenDoc**: Compound documents
- **Bento**: Object container system
- **Open Scripting Architecture**: Policies, protocols, and software for scripting
- **ComponentGlue Technology**: The OpenDoc-OLE interoperability technology
- **System Object Model**: The single machine version of Common Object Request Broker Architecture (CORBA), which includes multiprocess object invocation

CI Labs Mission

The form and function of CI Labs is similar to that of the X Consortium. Members of the consortium contribute toward the technical direction of the products. Members are allowed to modify and add value to source code reference implementations and to base products on this code. CI Labs will also play two other roles that have not been strongly part of the X Consortium. CI Labs will provide a certification service for developers. It will also provide a marketing role for its services and its technology base, along with other standards-based technology that becomes important to the goals of content integration, such as the technology from the Object Management Group® (OMG®).

Cross-Platform Coverage

Various companies will provide reference implementations for the CI Labs technology on the following platforms:

- **Macintosh**® (Apple)
- **OS/2**® (IBM)
- **Windows**™ (WordPerfect® from Novell®)
- **UNIX** (IBM)

Compound Document Architecture Concepts

Before looking at the specifics of the OpenDoc architecture, we will first consider the generic problem of designing a system to support embedding objects into a document and building component software. Most of this can be translated to both the technology of OLE and Taligent's document framework.

A fundamental concept of a compound document is that it can hold different types of data called *document parts*. Each document part is handled by an independent application or part handler. Each part handler understands its own intrinsic content, and even though other kinds of parts can be embedded into it, it need not know anything about the intrinsic content of that embedded part.

The integration and cooperation of the parts are handled by the architecture, policies, and protocols of the compound document system.

Other differences separate a document consisting of parts and a document created by a conventional application:

- **Content is not limited.** Users are not limited to the kind of content that can be put into a document. The only practical limitation is the availability of a part handler—an editor or viewer. As new part editors are acquired, they can be used immediately.
- **Content is built differently.** The user will rely more heavily on the clipboard and drag-and-drop mechanisms.
- **Part handlers can be changed.** Users can replace part handlers to fit their needs. This can be done independently of existing documents (which do not depend on a specific part editor).
- **There is a pervasive use of links for data transfer and navigation.** These links can be both internal to the document, such as hypertext type links, and external to the document for data transfer and navigation.

Compound Document Problems

Figure 4 uses a simple compound document to describe the problems faced in a compound document system and the interfaces for solving those problems. It takes the perspective of "If you were going to architect a compound document support library, what are the problems you would have to solve and the interfaces you would need to supply to the application developer?" Although there are several other problems associated with a compound document system that are not described here, these are the principal ones.

Data exchange—building the document. Many compound documents will be built up by the movement of data from one document (or the desktop) to another, either through cut-and-paste or drag-and-drop operations by the user. These interfaces determine how data is moved from one document to another. They must also define how data is put on and taken off the clipboard, along with passing type and attribute information so that the correct part handler can be accessed from the part handler database. Because the clipboard will be passing complex data objects, there must be an in-memory object container. The storage format for clipboard data will be a Bento container in the reference implementation of OpenDoc.

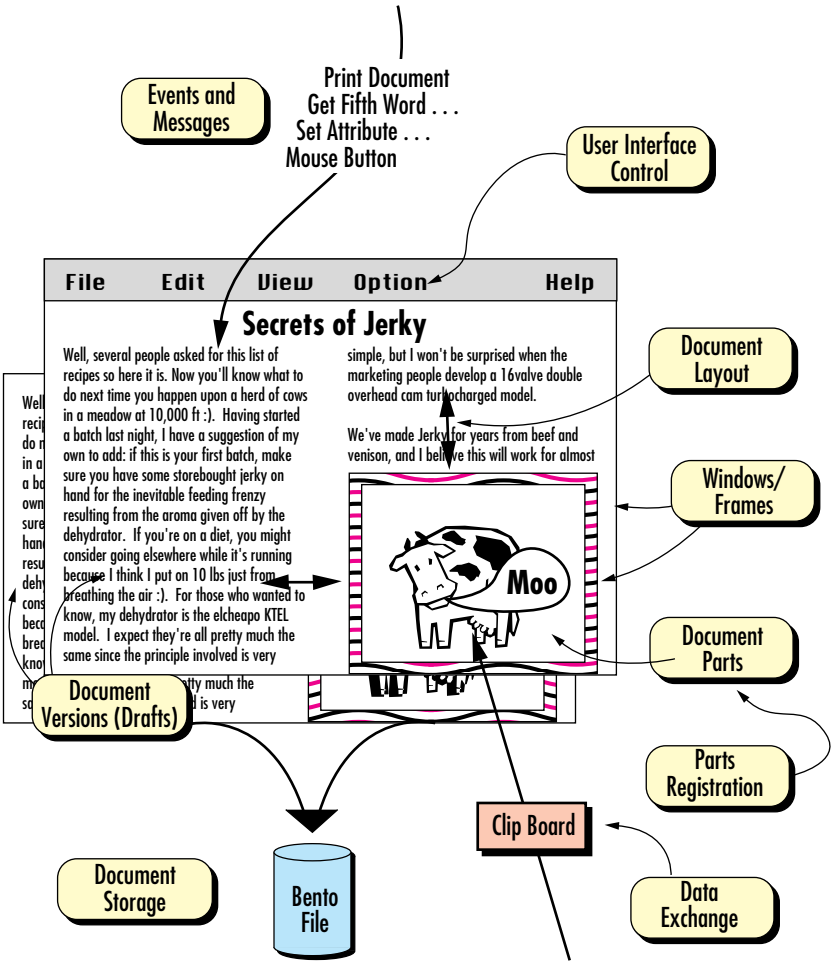


Figure 4. Generic compound document interfaces

Part registration—finding the right part handler for the data. When a container receives data of a particular type on the clipboard, it needs an interface to help it find the correct part handler for that type of data. Also, when a part handler is installed in the system, it needs an interface where it can register itself and the type of data (part types) that it supports. A part handler will often register itself claiming to handle several types of parts (such as different image formats), or there might be several part handlers that claim to handle the same part type.

For convenience, users can set up a preferences file to define which specific part handler is used for a particular type of part. An example of this would be a user who has two word processors on a workstation—MegaWrite and MuchoText. Although both have their own proprietary binary format, they also support the import of other formats, including the binary format of the other. The user could set up a part

OpenDoc Terminology

Term	Definition
Canvas	A surface to display frames in their facets. A canvas can be the display surface, a piece of paper, an off-screen display, and so on.
Container part	A part that can contain other parts. It has all the properties of a normal part, but it also tracks and responds to its embedded parts. The outermost or root container part is a somewhat special case. It takes up the entire document window and contains all other parts. An example of a root part would be a word-processing application. It has text as its native content, but can contain other parts to produce compound documents. These root parts often set the tone for document capabilities. A default root part contains only other parts and has no native content of its own. It simply provides a background.
Document draft	A version of a document in a single document structure. These drafts might be simple snapshots of the document as it is modified, or they might be drafts targeted for a specific audience (such as each draft localized to a language).
Event	There are two levels of events—low-level events such as mouse moves and key clicks, and high-level or semantic events, which are requests on an object to perform a function. The definition of the semantic events provides the basis for scripting.
Facet	The area on the canvas where the frame is displayed. Facets are not persistent—they are created and destroyed as frames are displayed and erased. For example, consider a multi-page document with many embedded parts. Parts that are not on the current displayed page do not have a facet. Parts that are displayed do have facets.
Frame	The area or border of a part. It encompasses the data in the part that is being viewed. It can be non-rectangular and displayed in one or more facets. In OpenDoc, parts can have multiple frames, with each frame representing a different view of the data. Frames are persistent and stored with the document.
In-place editing	The document-centric metaphor. When a user selects a part for editing or viewing, the edit can be done in place. A new window with the data need not be opened (although it can, if that is the policy of the part handler).
Link	A reference or pathway to an object. In compound documents, links are used in two ways: inter-document links such as hypertext links, and intra-document links that provide automatic updates of embedded information.
Part	The piece that gets embedded into the document. In OpenDoc, the embedded data is called the part, and the editor or viewer of that part is called the part handler. However, you will often hear the two combined with both the data and the editor/viewer referred to as a part. Conceptually, the part handler is an application program. Today, many applications, such as image editors, graphics editors, or audio players, will be modified to act also as OpenDoc part handlers.
Shell	The document shell and the root part are closely linked; however, it is useful to separate them functionally. The shell handles access to global information and the distribution of that information (by providing object references). It also receives low- and high-level events for the document.
Window	In OpenDoc, the outermost window of the document (the window for the root container). Other regions are called frames or menus, even though they might be implemented as windows in the native windowing system. This limitation in the use of "window" provides a cross-platform way of discussing OpenDoc regions.

handler preference with MegaWrite to be used for MegaWrite binary and MuchoText for MuchoText binary. This preference could be overridden if the user wanted to perform a conversion.

Documents—communicating between parts. If there are several embedded parts in a document, there should be an interface between parts and between the *containing part* and its embedded parts. Such an interface might provide the following.

- Access to internal part data
- Commands to the part, such as "save your data," "externalize yourself," and so on
- Requests to receive or give up control of some global resource, such as the menu bar or input focus

Geometry of part windows and frames—coordinating layout. With all of these parts sharing space on the screen and in the printed document, central coordination over layout is

necessary. This set of interfaces deals with the geometry of both the document window and the embedded parts. It contains information about the size, shape, position, clip extents, and transformations of the frames and windows, along with the means to set and manipulate them.

Document layout—determining the size. This interface negotiates space resources between the part handler and the container. It is used when the part needs to grow or shrink in size because of user interaction, such as editing the picture, or because the data changes after a link update of the data. The container, which sets the policy for document layout, has the final word. The result of this negotiation might be that the part is placed on a new page, the part is split into more than one piece, the document is rearranged to accommodate the new size, or even that the part is not allowed to grow.

User interface control—activating menus. This set of interfaces negotiates access to the user interface and other global resources. When a user selects a part with the mouse, the part handler needs to display its own user interface and menus to replace the previous ones. Additionally, there must be some policy set by the container for which menus are global to the entire document rather than just a selected part; for example, "print" under the "file" menu.

Events and messages—sending and receiving data. Several interfaces are needed for distributing, sending, receiving, and interpreting events and messages. These interfaces cover low-level events such as mouse moves or button pushes, middle-level events such as window manager events, along with high-level events or semantic messages. These interfaces are also the foundation for scripting, since scripts can be collections of semantic messages.

Documents—collaboration and versioning. These interfaces deal with multiple drafts or versions of a document. They handle the selection and creation of a draft, along with the management and reconciliation of multiple drafts. These interfaces are closely related to the storage interfaces.

Documents—storing and retrieving. These interfaces deal with the storage and retrieval of the document and its parts. They need to be abstracted from the underlying storage system so that a multiplicity of storage systems can be used. In the OpenDoc reference implementation, Bento containers will be used to build the storage system. It is likely that storage systems using OLE storage, Taligent storage, and object databases will be available.

OpenDoc Programmer Model

The programmer model for OpenDoc is a blending of function-based applications with classic object-oriented programming. OpenDoc was architected to allow easy conversion of existing applications to being OpenDoc part handlers. Externally, the application conforms to the OO interfaces of OpenDoc by using the `ODPart` object. Internally, the application continues to function as before. With Parts frameworks available, conversion can be straight-forward and rapid (see page 20 for some common OpenDoc terminology).

Class Library of Objects

In the previous section, the basic interfaces for a compound document architecture were generalized. In OpenDoc, all of the interfaces are realized as a class library of objects—System Object Model (SOM) or CORBA objects to be specific. Figure 5 shows the inheritance relationship of the OpenDoc class library.

The runtime relationship of the objects is more meaningful to the developer. Figure 6 shows the runtime relationship of the major OpenDoc objects. The shaded part of the figure represents the most important part for developers.

Part Developer Major Interfaces

The major programming task in creating an OpenDoc part editor is to subclass the class `ODPart` and override its methods. `ODPart` has 62 methods included in its interface. However, for simple parts and quick prototypes, only a dozen of these interfaces need to be changed. At a minimum, your editor needs to draw its part, retrieve its part's data, activate its part, and handle events sent to its part.

Since the editor will likely need a command or user interface to do this, it must provide menus for the menu bar as well as optional windows, dialogs, and palettes.

If the part editor will contain other parts, it must embed frames, create facets for visible frames, and store frames. The part editor can be extended to support asynchronous drawing, drag-and-drop and cut-and-paste, scripting, and links.

For converting an existing application, the simplest technique is to start with one of the provided sample parts and subclass it. This ensures correct default behavior for content-independent

The major programming task in creating an OpenDoc part editor is to subclass the class ODPart and override its methods.

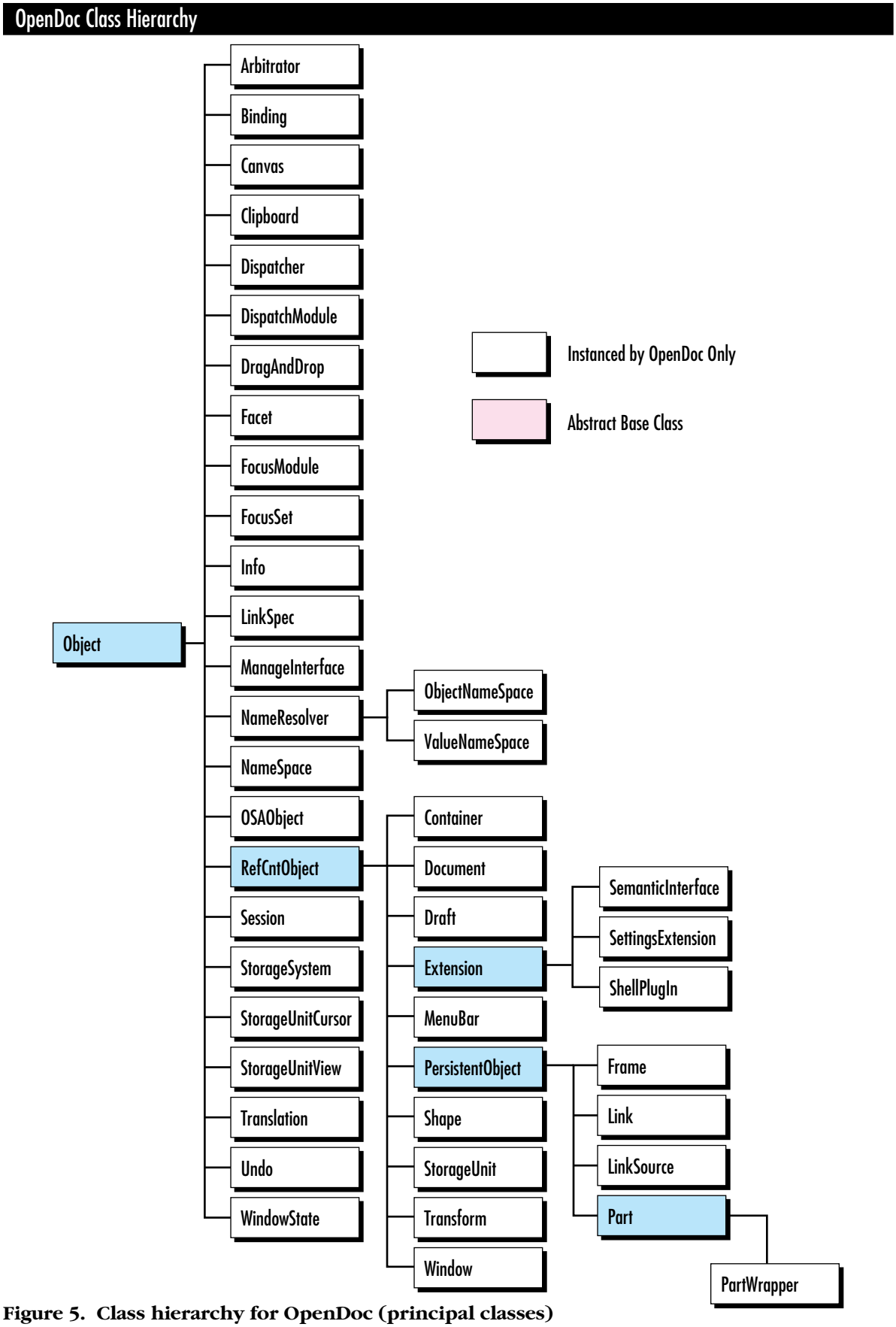


Figure 5. Class hierarchy for OpenDoc (principal classes)

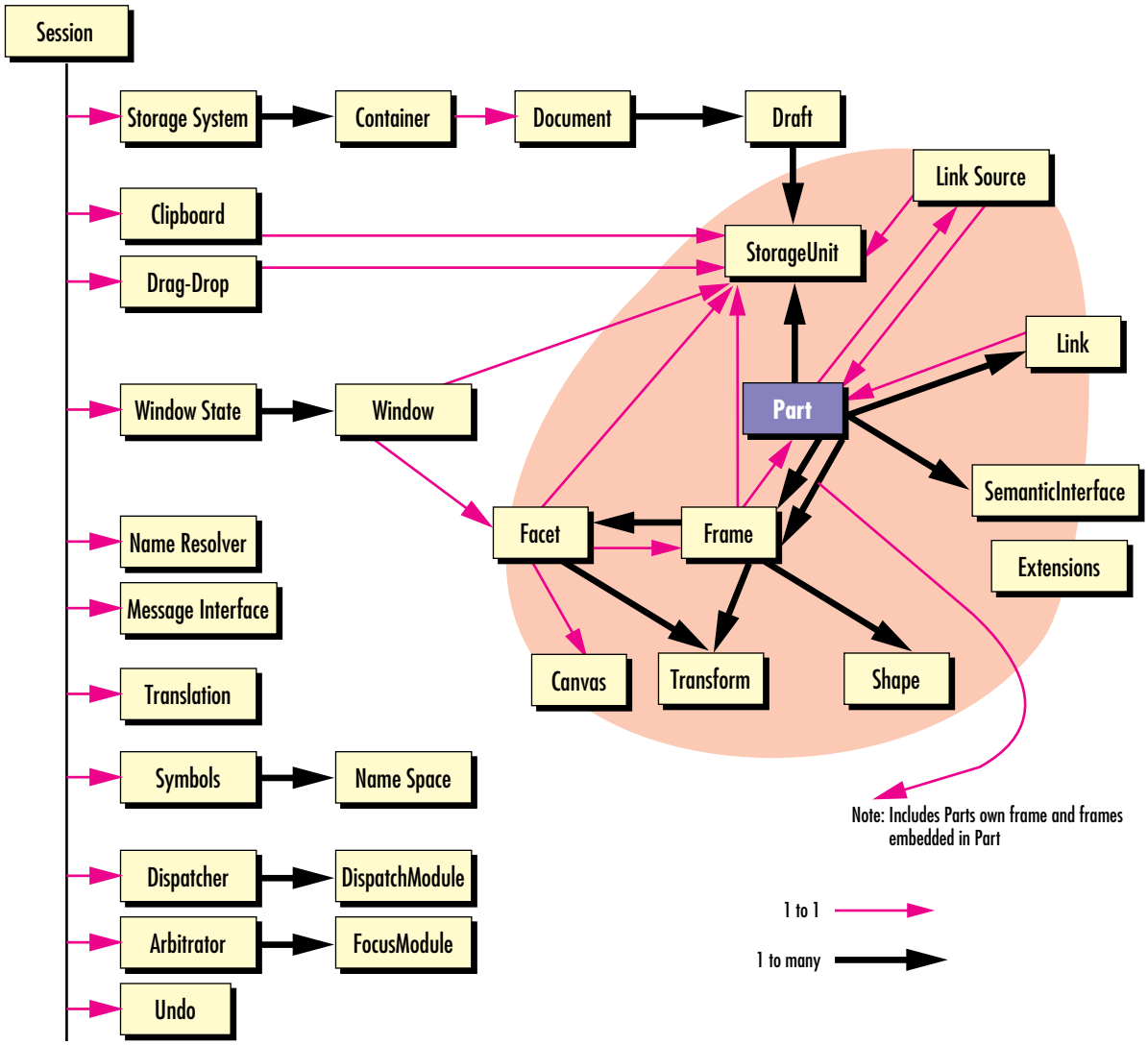


Figure 6. OpenDoc runtime object relationships

actions but allows for rapidly adding application function. In the past, OpenDoc developer labs (Parts Kitchens) have seen the majority of prototype applications converted in less than a week during lab times. Most of the platform vendors continue to offer Parts Kitchens.

Bento—An Object Container System

Bento is a simple piece of technology for containing objects (*Note*: The word container is used differently here than in OpenDoc). Think of it as a "Federal Express" envelope for objects. You can put anything into it, and there are established procedures for moving it around, looking at the inventory list, taking things out, and adding new things to the envelope—all independent of what is placed into it.

In OpenDoc, Bento has three primary roles:

1. A reference file storage system
2. An object container for items placed on the clipboard
3. Container of choice for transporting documents across platforms

You can think of Bento as a file system within a file; but rather than being a tree structure that reflects the embedded relationship, it is a flat structure. Figure 7 shows the structure of the Bento container with a series of objects. Each object has one or more properties associated with

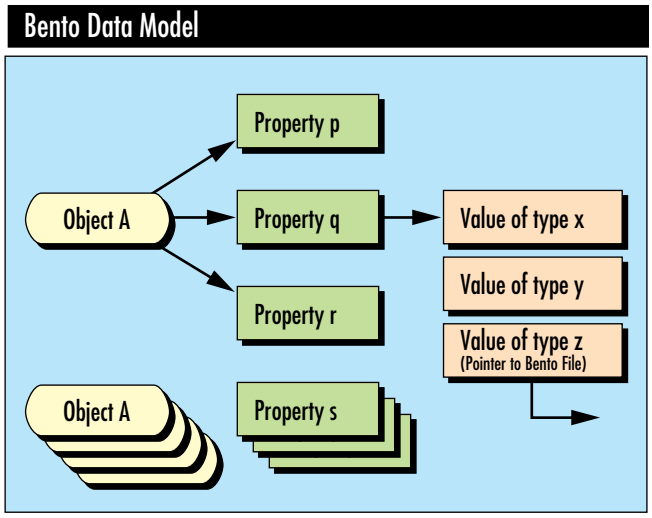


Figure 7. Bento data model

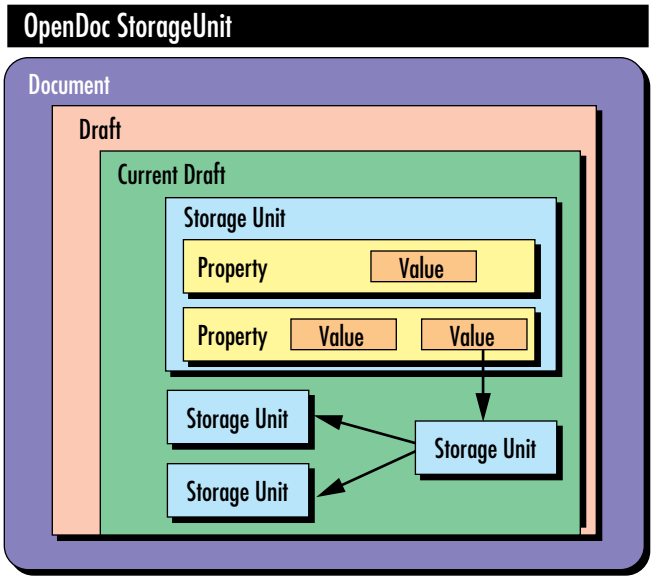


Figure 8. OpenDoc StorageUnit structure

the object, and each property has a set of values or data. In the example, property q might be formatted text stored in several different formats, such as RTF, SGML, and flat ASCII. Another valid value might be a reference to another Bento container.

Normally, the OpenDoc programmer does not deal directly with the Bento structure, but deals instead with the OpenDoc StorageUnit interface. The structure of the StorageUnit object is similar to that of Bento but can also be the front end for other types of storage such as an object-oriented database management system, or other object storage system such as the OLE Document File Format. Figure 8 shows the structure of the OpenDoc StorageUnit.

Open Scripting Architecture

The Open Scripting Architecture (OSA) provides an open architecture for scripting languages such as OREXX, AppleScript®, ScriptX®, or OLE Automation. The architecture is composed of two major components. The first component is the typing of script types, which allows the correct scripting engine to be included for script processing. The second component is the standardization of script semantic messages in the form of Open Events, which all OSA-compliant scripting languages will emit.

Scripts become part of the document, attached or embedded as needed. They can be the foundation for control structures such as front or back ends to operations, or they can be instructions that are embedded into a document that is mailed out.

Open Events

Open Events are based on standard registry of verbs and object classes. Apple Events provide the basis of this technology. Open Events are arranged in application suites, such as the following examples:

- Required
- Core
- Text
- Table
- Database
- Compound document
- New suites defined by user community

Each event is composed of three elements:

- **Events (the verbs):** Open, close, select, get, and put
- **Object classes or object specifiers:** Application, document, word, paragraph, and circle
- **Descriptor types:** Boolean, fixed, attribute greater than, 3rd, and bold

By combining these elements, single event messages of considerable complexity and richness can be composed. An example of a message for a text suite application might be the following:

```
Get all words that have the first character "w" and have the bold attribute in
paragraph 5 to paragraph 12 of the document called "foo."
```

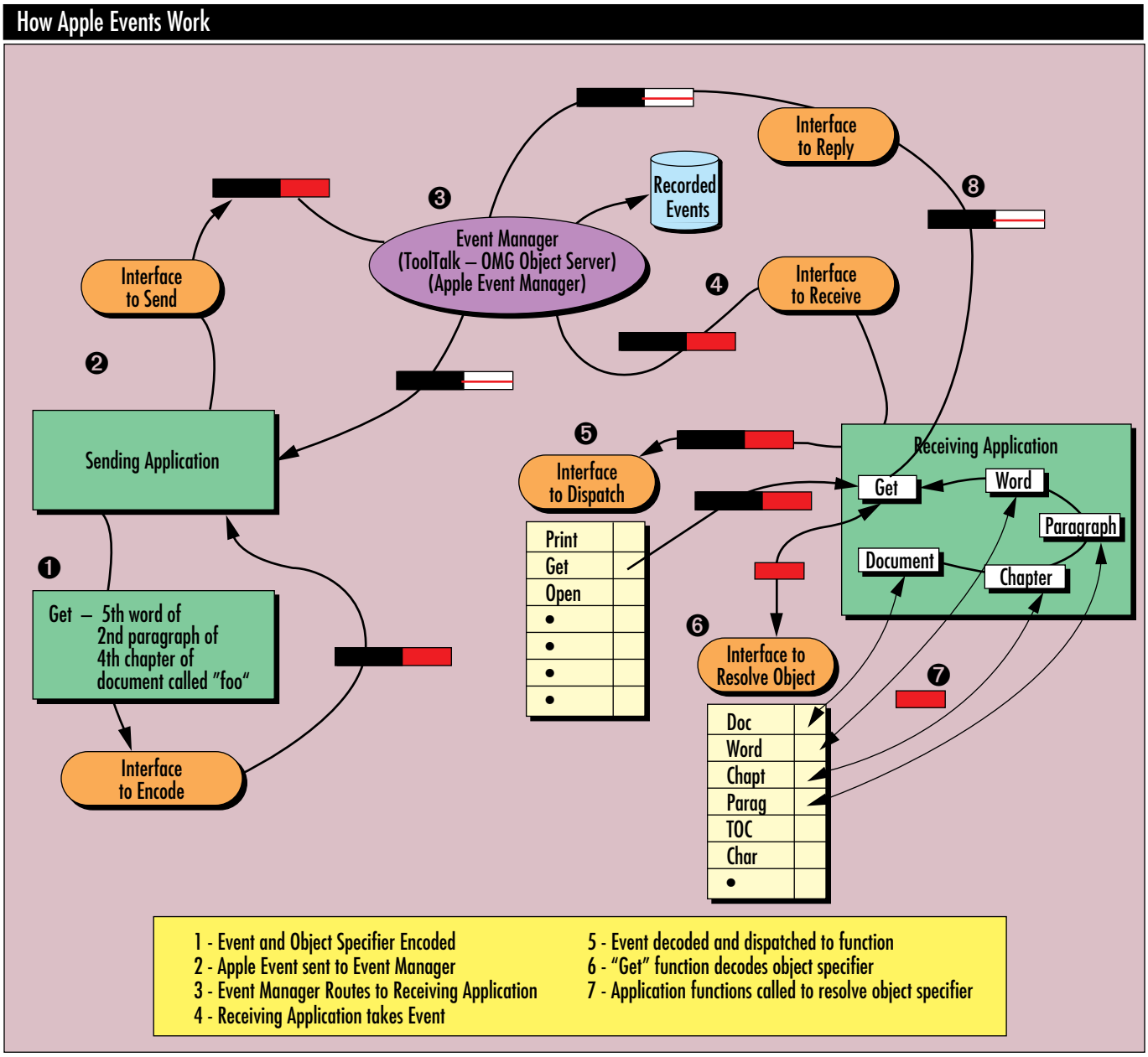


Figure 9. How Apple Events work

Parts that are OSA-scriptable register their capabilities by declaring the event suites that they support. Scripts can then be built for a particular set of suites, and those scripts will run on all parts that support those suites.

The set of event messages is open-ended. Parts developers can extend a suite's set of messages or define completely new suites. The extensions can remain proprietary or can be registered with CI Labs. The set of events for the suites beyond Required and Core is small. In most cases, the verbs from the Core suite are used with some of the semantics redefined. The object specifiers for each suite provide the real richness, and each suite has a hierarchy of objects on which it can act. The characteristics and attributes of each object specifier appear in the suite documentation.

Making a Part Scriptable

It is beyond the scope of this article to provide a detailed discussion on how to script an application or part, but some excellent documents and white papers on this subject are available via

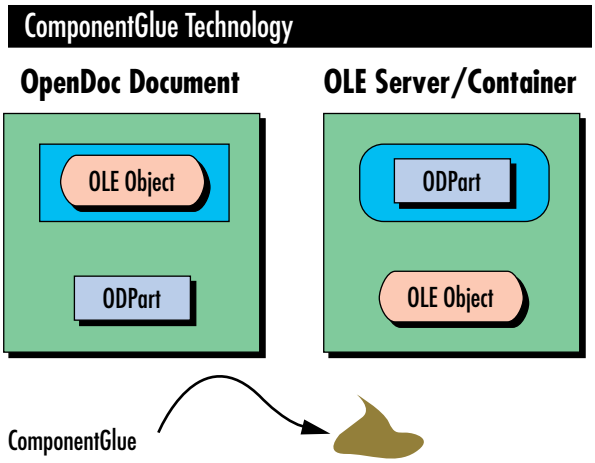


Figure 10. OpenDoc-OLE Interoperability—ComponentGlue technology

anonymous ftp directly from CI Labs (see the last section of this article for details).

Figure 9 shows the steps necessary for a script to be generated, transmitted, acted upon, and have information returned. There are three key points in this figure:

- OpenDoc has utility functions that help in building event records, delivering event records, and decomposing event records for action. For each of the event messages that a part supports, OpenDoc has registered a function that should be invoked—essentially a callback. This is also the mechanism by which it extends the set of event messages it understands. This is part of the semantic interface of OpenDoc.
- For each of the object specifiers and object descriptors that a part supports, OpenDoc has registered a set of functions that support them—essentially callbacks.
- Complex object specifiers are decomposed from outside-in or top-to-bottom. At each step, a reference to the acquired data is passed—first the file, then the chapter in the file, then the paragraph in the chapter in the file, and so on.

The example message `Get the 5th word of the second paragraph of the 4th chapter of the document called "foo"` perhaps looks somewhat contrived. But consider the following:

- The sending and the receiving applications might be the same application. The application has been factored into a UI component and a data engine component.
- The description of the example is a word chosen by a user and about to be put on the clipboard. Or perhaps it is a hypertext reference.
- The event message can be captured and stored (along with others) for future use.
- The stored events can then be fed into a scripting interface, generalized as needed, and turned into usable scripts.
- The developer's main task is to provide functions that support the event messages and object specifiers, and to factor their application into UI and data engine components.

System Object Model

SOM, IBM's implementation of the OMG CORBA standard, is an object-oriented technology for building, packaging, and manipulating binary class libraries. All interfaces for OpenDoc objects are written as CORBA Interface Definition Language (IDL) interfaces. SOM is one of the technology components that is part of the CI Labs OpenDoc technology, and all four of the current OpenDoc reference implementations use SOM as their Object Request Broker.

Features of SOM and CORBA

Some characteristics of SOM and CORBA are listed below.

Binary compatibility: SOM maintains binary compatibility between versions of class libraries. A binary class library such as OpenDoc shipped as a Dynamic Link Library (DLL) can be upgraded and changed without requiring a recompile or link of applications that use it.

Interface Definition Language (IDL) as defined by OMG: This is how CORBA-compliant objects are specified. All OpenDoc interfaces are defined in IDL, which has a similar look-and-feel compared to C++. IDL-specified interfaces are converted to a target language via compilers for the implementation of CORBA. SOM has IDL compilers for both C++ and C, with those for other languages such as Smalltalk® in process.

Language neutral: Because SOM is located in the middle of object transactions, objects written in one language can transparently make requests on objects written in another language.

Object Services: These services are needed to work effectively with objects. OMG is defining the interface and function of a set of services. Examples of Objects Services include life cycle—

create, delete, security, object naming, notification, and persistence. SOM has a framework for adding Object Services functions, so customers can select the capabilities they need and plug them in.

Object Management Group (OMG): A consortium set up to standardize an object architecture. The three main components of that architecture are an ORB, Object Services, and Common Object Facilities.

Object Request Broker (ORB): This entity brokers all requests one object makes on another. It is expected to transparently handle issues of location. It is also responsible for supporting object services such as creation/deletion and security. The OMG-defined ORB is called CORBA.

ComponentGlue Technology—OpenDoc and OLE

Technology being developed by Novell for CI Labs will allow OpenDoc and OLE parts to be seamlessly used in each others' documents by wrapping the parts with the appropriate object interfaces. The parts wrapper is called the ComponentGlue Technology or Component GT. This technology also supports the exposure of OSA-enabled part handlers as an OLE automation interface. Figure 10 depicts the ComponentGlue technology. A similar approach is being produced by IBM and Taligent for the Taligent document framework.

This ability to interoperate with both OLE and Taligent makes OpenDoc an ideal development platform for providing parts for all three architectures. It will allow developers to extend the life of existing software products with a quick port to OpenDoc. It also offers a migration path to providing Taligent-based products by incremental introduction of Taligent technology into the developers' software base.

Where to Get More Information

The best source of OpenDoc information is CI Labs. Documents, white papers, reference documents, and programmer guides that cover all of the technologies are available via anonymous ftp.

You can get additional information and subscribe to various CI Labs interest groups directly from the CI Labs mail server, cilabs.org. To access this server, do the following and you will receive a set of instructions via Internet mail.

1. Send a message to listsproc@cilabs.org
2. In the body of the note on a line by itself, put the word "help"

Information about developer CD-ROMs, Parts Kitchens, and beta programs is available from the following sources:

- **Macintosh:** opendoc@applelink.apple.com
- **Windows:** opendoc@wordperfect.com
- **OS/2:** 1-800-6DEVCON
- **UNIX (AIX®):** opendoc@austin.ibm.com

Chris Nelson, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Internet: cnelson@austin.ibm.com. Mr. Nelson is an architect in IBM's AIX Desktop group. He has a BA in Chemistry from Ripon College in Ripon, Wisconsin and a BS in Electrical Engineering from the University of Texas at Austin.