OpenDoc™

OpenDoc Technical Summary

By
The OpenDoc Design Team

Version 1.0
October 18, 1993

TABLE OF CONTENTS

SUMMARY

OpenDoc is the enabling technology which will bring a new class of applications and documents to the Macintosh, Windows, UNIX, OS/2, and other platforms. With OpenDoc, Apple, other platform vendors, and hardware and software developers will be in a better position to deliver new hardware and software technologies to users, to deliver better client/client/server integration to corporate users, and to deliver multimedia content to users with unprecedented ease.

OpenDoc itself enables a class of applications which can support compound documents, which can be customized, used collaboratively, and are available cross-platform. In doing so, it will fundamentally change the nature of software development for personal computers.

This document describes the requirements for the base OpenDoc technology: a set of libraries which developers can use to build compound document editors and viewers. The APIs for this base technology are extensive, covering hundreds of pages of material. This document may or may not be delivered with attached APIs. These APIs will be generally available at a later date.

INTRODUCTION

OpenDoc is an architecture for applications. Architecture is a word with many connotations, so we need to be a little more clear about our specific meaning of the term. Let's look at a definition for architecture from Webster's Dictionary:

> *Architecture (n.) A method or style of building. A unifying form or structure.*

To apply this same idea to software, we might end up with a definition as follows:

> *Application architecture (n.) A style of building applications, characterized by a particular program structure and user interface.*

The purpose of an application architecture is to create a style of building applications. A recipe, if you will, for putting together a fine program.

This document describes how to create an application which will meet the needs of users, developers, and platform vendors into the coming decade. OpenDoc applications provide greatly improved content integration, collaboration, customizability, and cross-platform interoperability.

Goals

OpenDoc has several major goals, each of which is equally important to the success of the package.

Compound document support

The major noticeable feature of OpenDoc is a set of interoperability protocols designed to allow code produced by independent development teams to cooperate to produce a single document for the end user. We provide APIs designed to allow these cooperating executables to negotiate about human interface resources, document layout on screen and on printing devices, share storage containers, and create data links to one another.

Scriptability

A second major feature is the ability to have parts be scriptable, and thus enabling end users to customize their applications to user-specific tasks. In a compound document environment, this capability allows sophisticated users to create simple applications using standard document editing procedures. It also allows programmers to disguise complex client applications as documents.

Collaboration Support

OpenDoc documents must have the ability to be passed through review cycles with minimal pain on the part of users. OpenDoc 1.0 implements a draft capability, which protects OpenDoc documents from data loss in cross-platform situations.

Cross-platform

OpenDoc was designed as a cross-platform architecture. To this end, we will be releasing OpenDoc as a reference specification and implementation. Commitments exist for implementations on AIX, OS/2, Windows, and Macintosh computers, with support for more platforms available soon.

Replaceability

To reach our goal of cross-platform adoption, OpenDoc must allow replacement of the implementation of any subsystem on any platform. Toward this end, we intend to fully document all meta-data for persistent storage so that interoperability is maintained.

Parsimony

The cross-platform arena is fraught with pitfalls. We have avoided attempting to specify drawing systems, coordinate systems, window systems, human interface guidelines, and many other platform specific elements. By doing so, we hope to make the architecture more generally available.

CHAPTER 1. BASIC CONCEPTS

OpenDoc is an architecture designed to enable the construction of compound, collaborative, customizable, and cross-platform applications. This segment of the document is designed to introduce you to the basic concepts of OpenDoc, to prepare you for the more detailed chapters to come later.

You should know that certain features have been designed as a part of OpenDoc which are not covered in this document. These include: authentication, digital signatures, store & forward messaging, compound user interface, and simultaneous access to documents. They are not covere here because they are not a part of the initial release.

Compound documents

OpenDoc enables the creation of compound documents. By this, we mean documents which are created and edited by several cooperating applications working within a single document.

Change to content centered model

OpenDoc fundamentally changes the meaning of the term *document*. In the present desktop computing realm, a document has a type which is used to choose the application which will help the user edit, view, and print the document. In OpenDoc, a document is a collection of *parts*, each of which is much like a present day document. Each part has a type, and this type is used to choose a *part handler* which will help the user edit, view, and print that part of the document.

The result of this shift in perspective is a significant change in how application software is written. The document is no longer a monolithic block of content, but is instead composed of many smaller blocks of content which together make up the content of the document. Since no single application has complete control of the document, protocols must be created to keep the various cooperating pieces of code from getting in one another's way. This is a pervasive change in how applications work. Protocols covering storage management, event distribution, run-time model, and human interface management must be followed if the document is to remain editable and uncorrupted.

This is a lot of work, but the rewards are great. Once OpenDoc is implemented, a whole class of existing user problems will go away. Users will be able to put any kind of content into any document without worrying whether the application can handle that kind of data.

Enabling compound documents

The central fact of life in a compound document world is that no single developer team is in control of a given document. The compound document integrates many different kinds of information, run by many different pieces of executable code, from many different developers.

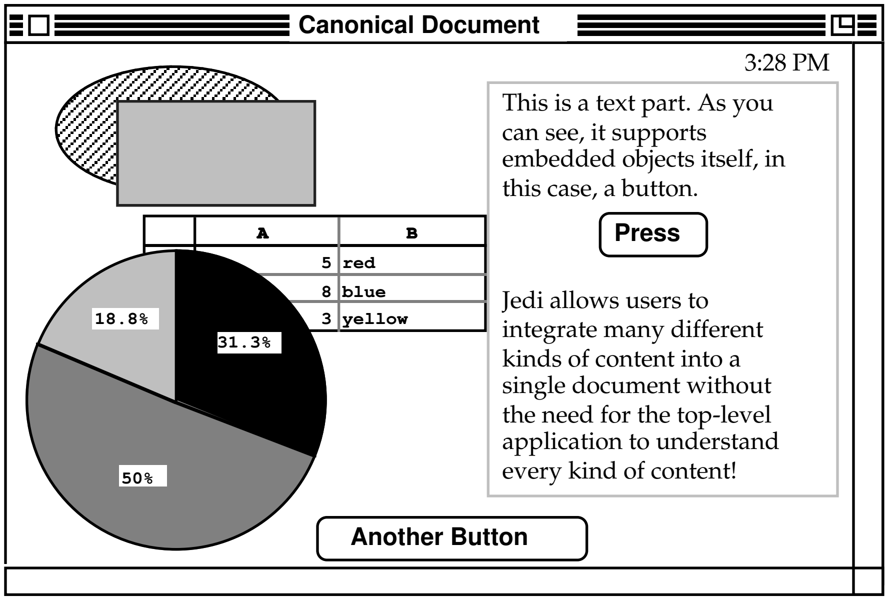
Parts

Parts are the boundaries in the document where one kind of content ends and another begins. Any document may have many different kinds of parts. These parts are a central notion in OpenDoc, since they give the user a way to predict what will happen, and how, when a change is made to a compound document.

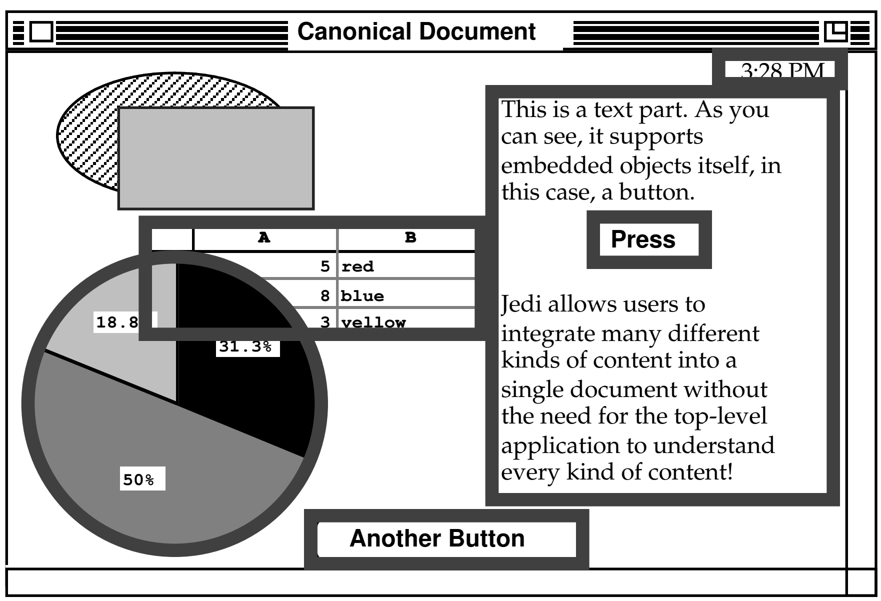
A part can embed another part, and know little or nothing about the information contained in that embedded part. The human interface team has constructed an interface designed to make it simple and convenient to select and operate on these embedded parts.

Every document has a single part at its top level, the *root part*, into which all other parts are embedded. Of course, some parts are embedded in parts which are themselves embedded in the root part. Transitively, though, every part is embedded in the root part of the document.

Here's an example document which we can use to illustrate the concept of parts.



In this example document, the root part is an object-style graphic editor. In the upper left corner, there are two content objects, a rectangle and an ellipse. A clock part has been embedded in the top right corner. Towards the center and bottom left, a chart part overlaps a table part. In the right center, a text part is embedded. A button part is embedded in the text part. A second button part is embedded in the graphics part, at the bottom center of the document. In the illustration below, a thick gray border is drawn around each part.



The key element of the notion of parts is that each part of a document has its own *content model*. By a content model, we mean the model of objects and operations that is presented *to an end user* of the part. The content model changes at the boundary between parts. Let's look at an example.

In the canonical document shown above, there is a text part. Inside the text part, the user sees lines, words, paragraphs, characters, and an embedded button part. Internally, the program in charge of the part might have a model which includes run-length encoding arrays for style information, line end arrays, and similar arcana. These are not presented to the user, though, so they are not part of the content model. In another part, the graphics part at the root of the document, the content objects may look very different. Circles, rectangles, and lines are content objects we would expect to see in a graphic part.

The content model also includes operations like selection, deletion, insertion, setting properties, and so on. Again, there are probably low-level internal routines, like piece table routines that allow efficient insertion of text. Just as with the objects, the ones considered part of the content model are the ones a user can access.

So we can properly define parts as bounded sets of content which have a single content model. Whenever the content model changes, a part boundary has been reached.

Notice that the rectangle and ellipse in the upper left corner of the example document are not parts at all, but are instead referred to as *content objects*. This brings up a key point. An embedded part is fundamentally distinct from the ordinary content elements like simple shapes, characters, cells, etc.. When a part boundary is reached, extra protocols must be invoked because more than one part handler is involved in giving the user access to the information.

Thus, in a working OpenDoc document, a part is both a user visible abstraction and a useful programmatic abstraction. A given part can be edited with minimal impact on the surrounding content of the document. Its storage can be managed independently of other parts.

Part handlers

Part handlers are the "applications" in the OpenDoc architecture. When a part is being displayed or edited, a part handler is invoked to perform those tasks. Just as an application performs basic editing, display, and printing tasks for a given document, part handlers perform these functions for a part. In terms of the content model we just talked about, the part handler provides the content model of the part.

A part handler is responsible for the following basic things:

• Displaying the part (both on-screen and printing). The part handler may be asked to display the part on a dynamic medium such as a screen or a static medium such as the printed page.

• Editing the part. The part handler must accept events and change the state of the part so that the user can edit and script the part.

• Storage management (persistent and runtime) for the part. The part handler must be able to read the part from persistent storage into main memory, manage the run-time storage associated with the part, and write the part back out to persistent storage.

A part and its handler together form the equivalent of a programmatic object, in the object-oriented programming sense of the word. The part provides the state information, while the part handler provides the behavior. When bound together, they form an editable segment of a document.

Part handlers are dynamically linked into the run-time world of the document, based on the part types that appear in the document. Because any sort of part might appear in any document at any time, the part handlers must be dynamically linked to provide a smooth user experience.

OpenDoc assumes that parts will mainly be used in a shell which allows a OpenDoc document to act like a present day application. This document shell will provide an address space, distribute events, and provide basic UI resources like windows and menus.

Editors and viewers

OpenDoc further breaks down the notion of part handlers into two concepts: part editors and part viewers. The difference between the two is that part viewers do not provide editing capability, but do provide full rendering and storage management. It is recommended that developers eventually create both kinds of handler for any given piece of content. The editor would be sold at an appropriate price, but the viewer could be freely distributable.

Events

Because one of the major aspects of OpenDoc is customization, OpenDoc part handlers must handle two distinct kinds of events: semantic events and UI events. The difference between the two types deals mainly with whether the event makes sense outside of the particular display context of the document.

A UI event needs information about where windows are located, how parts of the document have been scrolled, or in what part a selection was last made. These events include mouse clicks, keystrokes, and menu activations.

By contrast, a semantic event deals with the semantics of the document. Generally, they are independent of the graphical context, and deal with the content model of the part. Apple events which appear in the Apple event Registry are an excellent example of semantic events.

This distinction between the kinds of events yields a number of architectural advantages. OpenDoc applications are easier to script and record, but at the same time are easier to adapt to new user interface environments. OpenDoc documents and handlers are highly customizable as well, since the semantic interface allows extension or alteration of the UI without disturbing the core computational elements.

In OpenDoc, both kinds of events are passed to the part handlers by a generic document shell. This document shell performs many document wide functions, and one of the primary ones is the passing of events to part handlers. Typically, this shell will be provided by the platform implementor of OpenDoc. As with all parts of OpenDoc, this document shell is replaceable.

Registration

In an earlier section, we mentioned that a part handler is bound to a part based on the type of the part. To perform such a binding, a table of which handlers go with which parts must be maintained somewhere in the system. We call this table the part registry. There are a number of similar tables which are associated with areas like data translation and scripting systems. In each case, the table is used by a binding mechanism to choose the proper code to execute a task.

In considering how to perform registration and binding, we considered two fundamental aspects: user preference and document stability. Both are fundamentally user experience issues. The user wants documents to remain stable in content and behavior no matter where they are moved. At the same time, the user wants his or her chosen set of part handlers to remain stable as well. This is a fundamental tension, and we resolve it by providing a series of behaviors which we feel provide a good mix of both desired responses.

Storage

Storage is a major issue in OpenDoc. Given the presence of multi-part documents, a persistent storage mechanism must be created which enables multiple part handlers to share a single document file effectively.

OpenDoc assumes that such a storage system can effectively give each part its own storage stream, and that reliable references can be made from one such stream to another. Because many pieces of code may need to access a given part, the storage system must support a robust annotation mechanism to allow information to be associated with a part without disturbing its format.

Run-time

At run-time, OpenDoc assumes that an instance of the document shell will be created for each document. This generic shell will be responsible for providing four basic structures to the part handlers. These are: the storage system, the window and its associated state, the event dispatcher, and an arbitration registry to allow negotiation about shared resources like the menu bar.

On the Macintosh, the run-time shell of a OpenDoc document will also be responsible for binding and loading part handlers for the parts which appear in the document. It is assumed that once a given part handler is loaded, any part in any document may share the part handler's executable code.

Layout

Once several parts are competing for space on the display or the printed page, layout can become complex and difficult. As a result, OpenDoc defines a concept called a frame to regularize the negotiation between parts for space.

Frames form the basis by which documents are composed from smaller parts. The structure of the frames provides information about layout, containment, and document context.

Customization

Another major goal of OpenDoc is to increase the level of customization that a user can apply to his documents or applications. To achieve this goal, OpenDoc includes a notion of pervasive scripting and a notion of compound user interface.

Scripting

Content centered scripting

OpenDoc includes a notion of content centered scripting. By this, we mean scripting based on the content model of parts. To be scriptable, a part handler must have a content model, with its constituent lists of content objects and operations. The OpenDoc model assumes a method of delivering semantic events from a scripting system to part handlers, which respond to the events. Apple events is an example of such a system.

To reach higher levels of scripting support, more support is needed from the part handler. There are a series of possible options: simple scriptability as described above, tinkerability, and recordability. Each kind of support provides useful features, at some implementation cost.

Compound document scripting

OpenDoc allows the scripting of compound documents. By specifying a naming scheme that can span part boundaries, the user can navigate to the content object in question using familiar document structures. Any part of any document visible on the network can be scripted transparently. A given semantic event, however, cannot act in more than one part at a time.

Compound UI

OpenDoc supports a notion of compound user interfaces. This means that one can use OpenDoc parts to build user interface elements like dialogs, tool palettes, tool ribbons, and menus. By using scripting to let these UI elements and the part handlers communicate, end users or in-house development shops can alter or extend the graphical interface of the part handlers they purchase.

This capability is allowed, but not implemented, by OpenDoc. A future project may provide the foundation for creating this kind of user interface in conjunction with OpenDoc.

Collaboration

Collaboration takes many forms in OpenDoc. The word collaboration means many things to many people. It may mean working together on the same content, or exchanging information between differing content environments.

OpenDoc provides recipes that enable the other kind of collaboration, that of exchanging information between OpenDoc documents programmatically. Lastly, OpenDoc provides mechanisms that allow users to collaborate safely, protected from malicious or accidental alteration of their information.

Mail

One advantage of OpenDoc is the separation of part handlers from document level functions. This places Apple and other platform vendors in an excellent position to add new features to documents without asking developers to revise their applications. OpenDoc's document shell thus provides the access to platform mail facilities, particularly mail sending, without developer interaction. However, part handlers with knowledge of specialized mail or messaging systems will still be free to implement features based on those systems.

Collaboration between parts

OpenDoc allows for collaboration between parts using scripting. Scripting forms a rich medium for coordinating the work of parts in documents, and allows users and parts to work together to perform tasks. By extending scripting to use messaging services as well as direct calls between applications, collaboration over time or space becomes available to smart documents as well as users.

In addition, OpenDoc supports a notion of extended interfaces between parts. These interfaces extend the basic OpenDoc interoperability layer with specific added functions which developers can use to increase collaboration between parts.

Shared documents

Another major area of collaboration is the shared creation of documents. OpenDoc places great emphasis on enabling users to collaborate through a mechanism familiar to many people: *drafts* of a document.

When multiple users share a document, each can work in his own draft, which can be reconciled manually with other user's drafts at a later time. Users can look back through the drafts of the document they and others have created.

Drafts are document-wide, and store only those parts of a document that have changed in each draft. As a result, it will be practical for users to store several drafts in a document without prohibitive space overhead.

Document interchange

Needless to say, when people are sharing documents, the issue of data translation will arise. OpenDoc provides access to translation in a number of areas: clipboard transfers, document transfers, messaging, and part instantiation.

When people are sharing documents, data integrity becomes a major issue in a component world. If users have different part handlers for the same parts, subtle difficulties can arise as data is passed back and forth between users and their machines. Differences in display geometry, indexing schemes, or I/O implementations can cause slight but annoying differences.

OpenDoc helps solve this problem, again using the idea of drafts. If translation occurs in the process of sharing documents, the user can always consult an older draft to regain access to formatting information that might have been lost during document translation. OpenDoc documents can support multiple representations of translated documents.

Another aspect of maintaining data integrity is the promotion of standard formats for publishing a variety of different kind of content. OpenDoc provides a strong foundation on which these standards can be built. By providing a basic usage model for compound documents, OpenDoc enables content formats designed to work well in compound documents with extensive embedding.

Cross-platform

OpenDoc is an architecture which is intended to span platforms. We have, therefore, attempted to capture our assumptions about which platform features must exist, and what constraints are placed on their operation.

We have attempted to make our APIs general enough to adapt to most platform human interface and runtime models. We attempted to consider Macintosh, Microsoft Windows, Motif, OPEN LOOK, and OS/2 when considering how to build OpenDoc.

Issues of data integrity are also critical in cross-platform settings. OpenDoc will be promoted by an organization devoted to solving these issues in a vendor-neutral fashion. This organization, the Component Integration Laboratories (CIL), is composed of a number of platform and application vendors with a common interest in solving these issues.

CHAPTER 2. CLASS LIBRARY

OpenDoc is a cross-platform architecture for compound documents which can be used collaboratively, and can be scripted. This chapter provides an introduction to the classes implemented in OpenDoc's libraries, and the design goals behind them.

OpenDoc is built as an object-oriented system, but is designed to allow procedural code to be easily plugged into its overall structures. A key design goal was to allow existing applications to be retrofitted easily to work in an OpenDoc environment. Another design goal, that of replaceability, makes OpenDoc a level playing field on which many developers, at both the system and application levels, can innovate. Lastly, the system is extensible to assure a long lifespan.

This document does not describe the operation of the classes in detail, nor does it attempt to convey a sense of how the overall architecture was designed. Instead, it describes the class structure and object interactions of the first release of OpenDoc technology.

Design goals

Cross-platform

OpenDoc is designed to be a cross-platform architecture, easily adaptable to any platform both in terms of runtime model and human interface specification. It makes very few assumptions about the underlying platform.

Its primary assumptions are the following:

- There is an event based graphical user interface model for the platform.
- There is a method of dispatching messages between separately compiled executables. For encapsulation reasons, we express the boundaries between executables in terms of objects and methods.
- There is a persistent storage system with stream based I/O available.

Easy Retrofit

OpenDoc is designed to have existing code moved into it in as easy a fashion as possible. Thus, the design attempts to specify as little as possible about the internals of part handlers, the equivalent of existing applications in the OpenDoc world.

- Memory recovery model does not specify a particular allocation model or memory management scheme.
- I/O subsystem provides stream storage interface, and does not require a particular language or object format.
- Imaging subsystem does not require any particular drawing library, and attempts to constrain the drawing as little as possible.
- Event handling is designed to be as close to existing systems as possible, and includes support for modal behavior as well as edit-in place behavior.

Replaceable

OpenDoc is designed for its various sub-sections to be replaced by platform vendors as well as other software developers. To achieve this, we take several precautions:

- OpenDoc objects do not expose data members to outside view or manipulation.
- We define groups of objects which may be implemented as a set, called *cliques*. Cliques of objects often have private interactions amongst their members which are not publicly available. A clique must also have the characteristic that members of the clique are instantiated under the control of other members of the clique, except for a single master object class. One should not implement private interfaces which are used

beyond the boundaries of a clique. Every sub-system of OpenDoc is, by definition, a clique. However, in some cases, particular sets of objects within a sub-system are also defined as cliques.

Extensible

OpenDoc does not cover the entire space of possible component interactions. Far from it, it instead concentrates on human interface interactions such as event handling, scripting, and layout. However, we realize that it's quite likely that others will wish to extend the interaction space by adding new methods to the classes involved. We formalize this as follows:

- Every significant class has an extension mechanism built-in, which allows clients of the object to ask for named extensions.
- It is always allowable for a class to refuse to give a client access to an extension.

Not a framework

While OpenDoc is designed as a set of interacting objects, it is not an object-oriented framework in the traditional sense of the word. Specifically, the purpose of OpenDoc's design is not to make it *as simple as possible* to create compound documents, but merely to make it *possible*. This means that we concentrate on distribution of the necessary information, not enforcing consistency of interface or implementation.

Sub-systems

OpenDoc can be considered as a series of sub-systems, designed for easy replacement. Each sub-system has a set of associated classes, although some classes could be considered members of more than one sub-system. Nonetheless, the sub-systems are a good way to conceptualize the system at a high level. The sub-systems are: Shell, Storage, Imaging, HI Events, and Semantic Events.

Shell

The outermost sub-system, and by far the most platform-specific, is the shell. This system is not specified as a part of the basic OpenDoc design because its process model and human interface are very platform-specific. The shell is responsible for providing the process in which the Session object resides. It is also the focus of human interface interaction for the document.

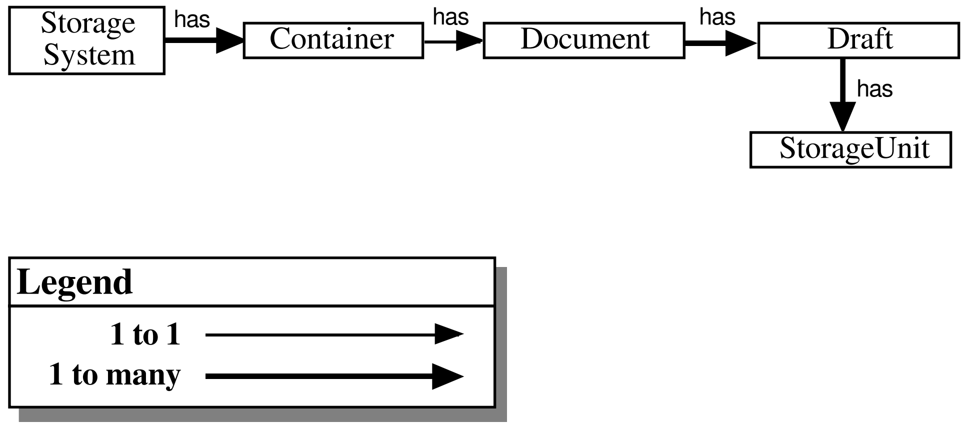
The shell's basic responsibilities are to:

- Create and initialize the Session object and its direct contents.
- Open the appropriate document (as instructed by the user using the platform human interface) from the storage system.
- Accept human interface events and pass them to the OpenDoc Dispatcher.

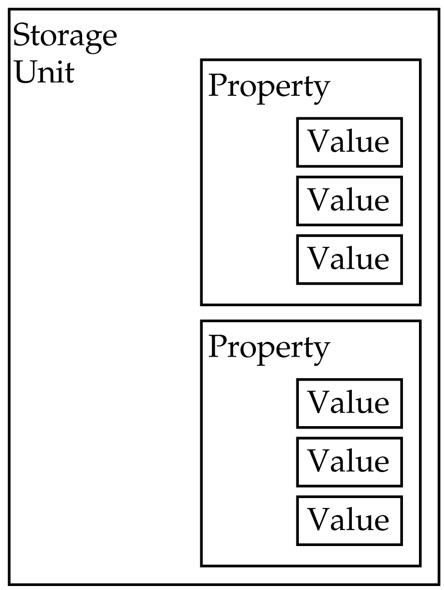
Storage

The storage system is the place where persistent storage is managed by OpenDoc. It is not an object-oriented database, but is instead a system of structured files, each of which contains many streams. It was designed this way to ease the transition for developers working with existing code bases, which generally assume stream based I/O.

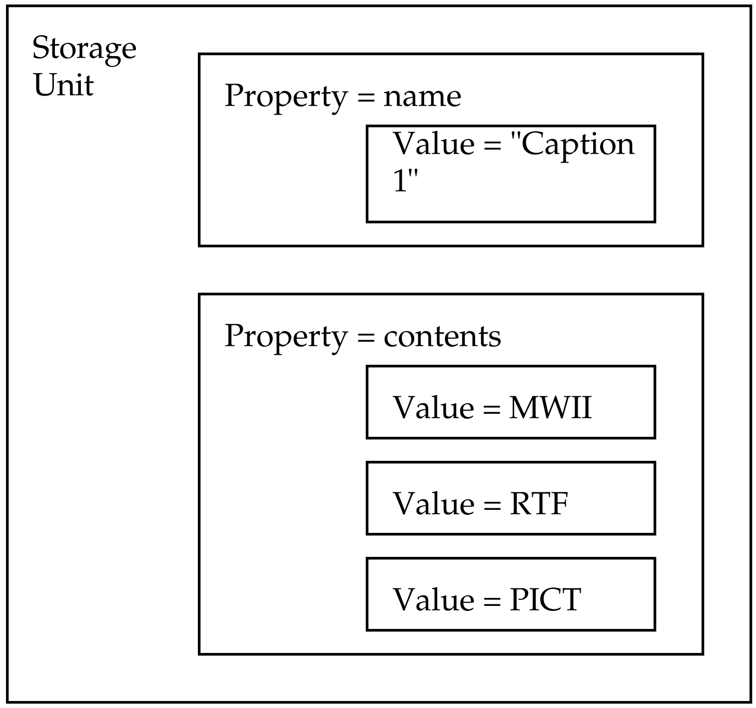
The primary object here is the StorageSystem object, which instantiates and maintains a list of Container objects. Each Container may contain a number of Document objects, each of which contains one or more Draft objects. Each Draft contains a number of StorageUnits, which are much like a directory structure in an existing file system. Every StorageUnit can be thought of as a list of properties, each of which has a number of streams called values. There can be several values for any property, and values are distinguished by type.



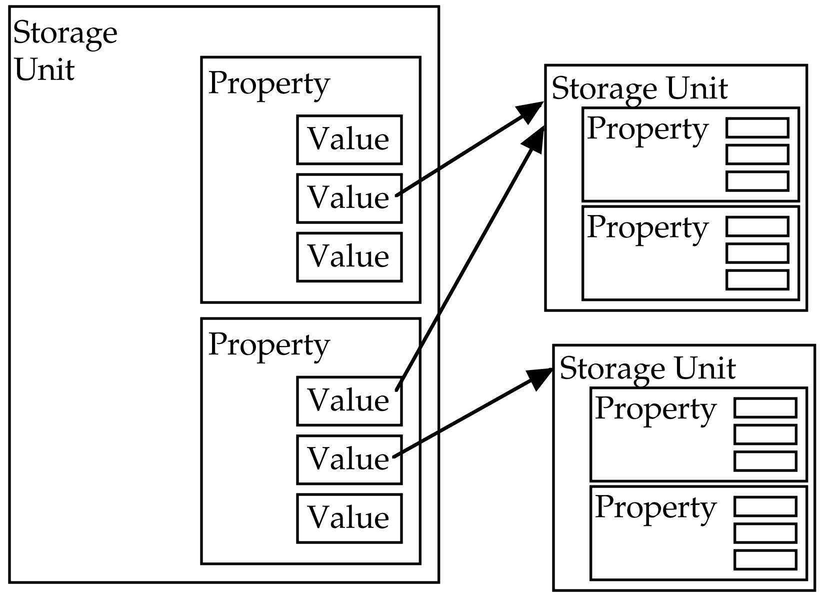
Here's a diagram of the contents of a storage unit, as well as a simple example:



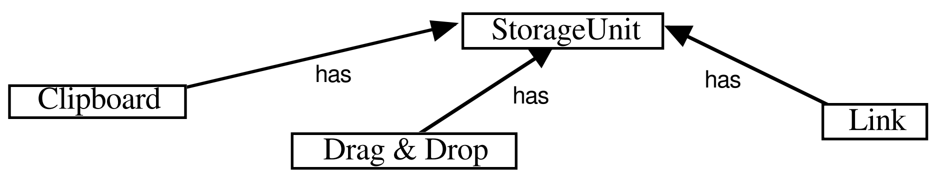
Okay, now here's an example of a StorageUnit in use. It is not a complete description, but serves to illustrate the use of a storage unit. This particular storage unit contains a bit of word processing content. The part involved is storing a name property as well as the contents property, where the primary information about the part is stored. This particular storage unit has a single representation of the name property, but has saved three representations of the contents property: a MacWrite II format, and RTF format, and a PICT format.



A storage unit can have persistent references to any other storage unit, stored in any of its values. This means that storage units can be arranged in many different organizing schemes. One typical organization is a hierarchy, and in fact this is how OpenDoc stores its embedding hierarchy within a document.



Also associated with the storage system are objects to support linking, the clipboard, and a drag & drop interface. Each of these classes makes a storage unit available for storing data to these data transfer mechanisms. In turn, each of the storage units is associated with a full draft for storing compound information.

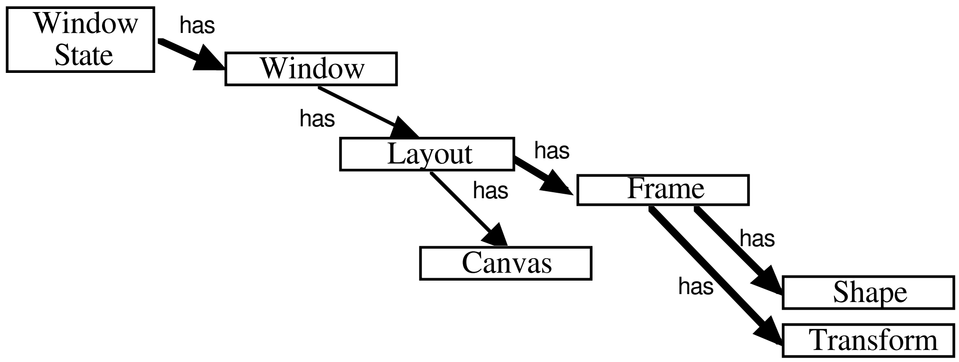


Each of these data transfer objects also has specific methods associated with its particular mode of data transfer.

Lastly, a pair of handy objects for referring to particular locations within a storage unit: StorageUnitView and StorageUnitCursor, provide easy and reliable access to particular values within a storage unit.

Imaging

The imaging subsystem is really a geometry subsystem, not a full imaging model. Its purpose is to provide a repository and protocol for describing part geometry in a document, whether for printing or window display.



The system derives geometry from three primitive notions, the implementations of which are platform specific. These are:

- Canvas: a drawing context which supplies a color table, coordinate space, etc.
- Shape: a description of a geometric area of a Canvas. In QuickDraw, this would be a region plus a 32 bit offset.
- Transform: a geometric transformation specific to the drawing system. For QuickDraw classic, this would be a simple offset. In more complex systems, such as QuickDraw GX, a full 3x3 matrix is a likely representation.

The actual process of imaging is not addressed by OpenDoc. That is to say, we do not have a part of the API which actually draws lines, circles, bitmaps, etc. Instead, we expect the part handler to make the appropriate calls based on the kind of Canvas onto which they are drawing.

Window State is merely a list of windows in which OpenDoc parts are displayed. The Window class represents a platform specific window, and holds a Layout object which specifies the relationship between various part Frames in that particular window.

The Layout class, just mentioned, handles the structure of a graphical layout. It holds containment and z-ordering information about a set of part frames. This data structure can be used to lay out a window's contents, or to lay out contents on any drawing canvas. Thus, the same Layout object can be used for printing as well as active display. To handle the difference between layout to a static device like a printer and an active device like a computer display, a flag is stored in the Canvas object associated with each layout.

The Frame objects contained within a layout are data structures designed to store geometry information resulting from interaction between a containing part and the embedded part being displayed in the frame. The frame handles update notifications and acts as a repository of geometric information.

Typically, during the imaging process, a part handler is asked to draw a particular Frame. The part handler gets the clipping, transformation, and layout information from the Frame, and then calls the desired graphics toolbox calls to perform the actual drawing.

HI Events

HI events are handled by two primary objects, the Dispatcher and the Arbitrator. Together, these classes deliver events to the appropriate part handler.

The Dispatcher is intended to accept HI events from the underlying operating system and dispatch them to part handlers. Typically, a part handler is unconcerned with the dispatcher, and is called by it at run-time. It is a modular design, which can be extended to handle new classes of events at run-time by adding DispatchModule objects. Each of these modules is responsible for reading information stored in the Arbitrator, Window State, or similar structure and deciding which frame, with the attendant part, should be asked to handle the event. Should the chosen part be unable to handle the event, it's container can register interest in the events not handled by an embedded part.

The Arbitrator is designed to arbitrate ownership of shared resources like the menu bar, keystroke stream, etc.. Like the Dispatcher, the arbitrator is modular, and can be extended to arbitrate software or hardware resources defined by part handlers or platform capabilities. The Arbitrator uses a two-phase commit scheme to avoid resource deadlocks, and can be used for distributed resources as well as local resources.

Semantic Events

The Semantic events sub-system is the first of a number of extensions possible under OpenDoc. The purpose of this extension is to allow OpenDoc parts to be scripted, meaning that they can be controlled remotely to accomplish some useful task. These tasks are specified using user-level abstractions, rather than programmer level abstractions. Semantic events are not an RPC mechanism, but rather a verbally intuitive command mechanism which can be used to automate user-level tasks. Several objects are involved.

The basic process of handling semantic events is that of accepting an event and then calling the part handler to invoke the correct code. Because semantic events generally need to be synchronized with user generated events to some extent, the Dispatcher implements the code to actually process the event and dispatch it to the appropriate part.

Because users must often act on lightweight objects which are presented by a part, we have designed a system for naming both lightweight and heavyweight objects. We call these names object specifiers. We also implement an object, the NameResolver, which takes object specifiers and resolves them into particular objects on which events can operate. Typically, an event arrives with one or more object specifiers, and the event handling code resolves these specifiers using the NameResolver, and then acts.

Both the event handling code and the code necessary to resolve object specifiers is exposed to the other parts through an extension object called the Semantic Interface. This is mainly a repository for code which can be called by the NameResolver and the Dispatcher.

Lastly, should part handlers wish to send semantic events themselves or make them available for recording, an object called the MessageInterface allows them to send semantic events and record them.

Note that the entire semantic events system is a pure extension to the basic OpenDoc system. Any object which does not publish a Semantic Interface object will not be scriptable, but will be a first-class citizen of OpenDoc in all other respects.

Class Hierarchy

The existing OpenDoc class system as it might appear in a C++ specific world is shown in Figure 3.1. Later in the development cycle, this hierarchy will be replaced by a System Object Model (SOM) based hierarchy and use abstract base classes with multiple inheritance to define the behavior of objects. At present, there is little inheritance in the hierarchy and extensive use is made of delegation instead. This preserves the language-neutral flavor of OpenDoc and prepares us to use SOM in the future.

The class hierarchy diagram does not contain every class defined by OpenDoc. There are a large series of iterators which do not appear, as well as one object which represents a simple set of other objects. These objects are simple to conceptualize, and are basically support objects for classes which appear in this diagram. We have left them out to aid initial understanding. Examination of the specific class APIs should make the purpose of these objects obvious.

The run-time state of an OpenDoc document, and the relationships which various OpenDoc classes have to one another is shown in Figure 3.2. The relationships generally consist of pointer references from one object to another. The labels are meant to give some human readable meaning to the pointer reference.

As has already been mentioned, the inheritance hierarchy of the classes is quite flat (Figure 3.1). This is because the team chose to use delegation as much as possible for maintenance and replaceability. There are three abstract base classes, for reference counting, extension, and persistence. Conceptually, these three capabilities are orthogonal. In practice, with the existing C++ based API, all ref-counted objects are extensible, and all persistent objects are ref-counted.

We will examine the classes in terms of the abstract base classes, followed by the various sub-system groupings, followed by a last category of "service" classes, which serve as utilities for the other classes to use.

Abstract Base Classes

RefCounted Objects

RefCounted objects are objects which maintain a count of how many other objects refer to them. The major object classes of OpenDoc are all RefCounted, so that session global memory can be managed. Objects cannot be removed from memory unless their reference counts go to zero.

Extensible Objects

Extensible objects are objects which can return extensions of themselves. These extensions are themselves objects, and are all RefCounted objects. An object which has special behavior beyond that specified by OpenDoc should make this behavior available through an extension. All subclasses of XMPObject are extensible.

Persistent Objects

Persistent objects are objects which have a designated StorageUnit object which is created when these objects are added to a storage draft. A small set of very important document structure classes is derived from this base class. All persistent objects are reference counted.

Implementation Classes

Storage

The primary objects associated with storage are StorageSystem, Container, Document, Draft, and StorageUnit. The StorageSystem object is designed to support Container objects which have any of several different implementations. Implementation of Container, Document, Draft, or StorageUnit will probably only work with other objects which were implemented as a clique. The Container class serves as the master object of the clique, since once a container is instantiated all of the documents, drafts, and so on are created by a member of the clique.

The data interchange objects: Link, Drag & Drop, and Clipboard, should be replaced as a clique along with whatever storage implementation they will be using.

Imaging

Canvas, Shape, and Transform are designed as a clique. Thus, they should be implemented together and should also handle translating between any graphics models specified by the platform implementor.

WindowState, Window, Frame and Layout are also a clique, and should be replaced together.

HI Events

The Arbitrator and Dispatcher are generally not replaced, and are designed to have their specific behavior code modularly replaced. Thus, there is no particular notion of clique membership involved in altering their behavior. However, there is some semantic event interaction with the Dispatcher, as noted in the following section.

Semantic Events

The NameResolver, SemanticInterface, MessageInterface, and the Dispatcher module associated with scripting events or synchronization between script execution and HI events should be replaced as a clique.

Service Classes

A few object classes appear mainly as services for other classes to use. The Translation object, for instance, is the API to platform specific translation services which may be available. The Symbols and Name Space classes exist to provide a handy implementation of attribute/value pair storage.

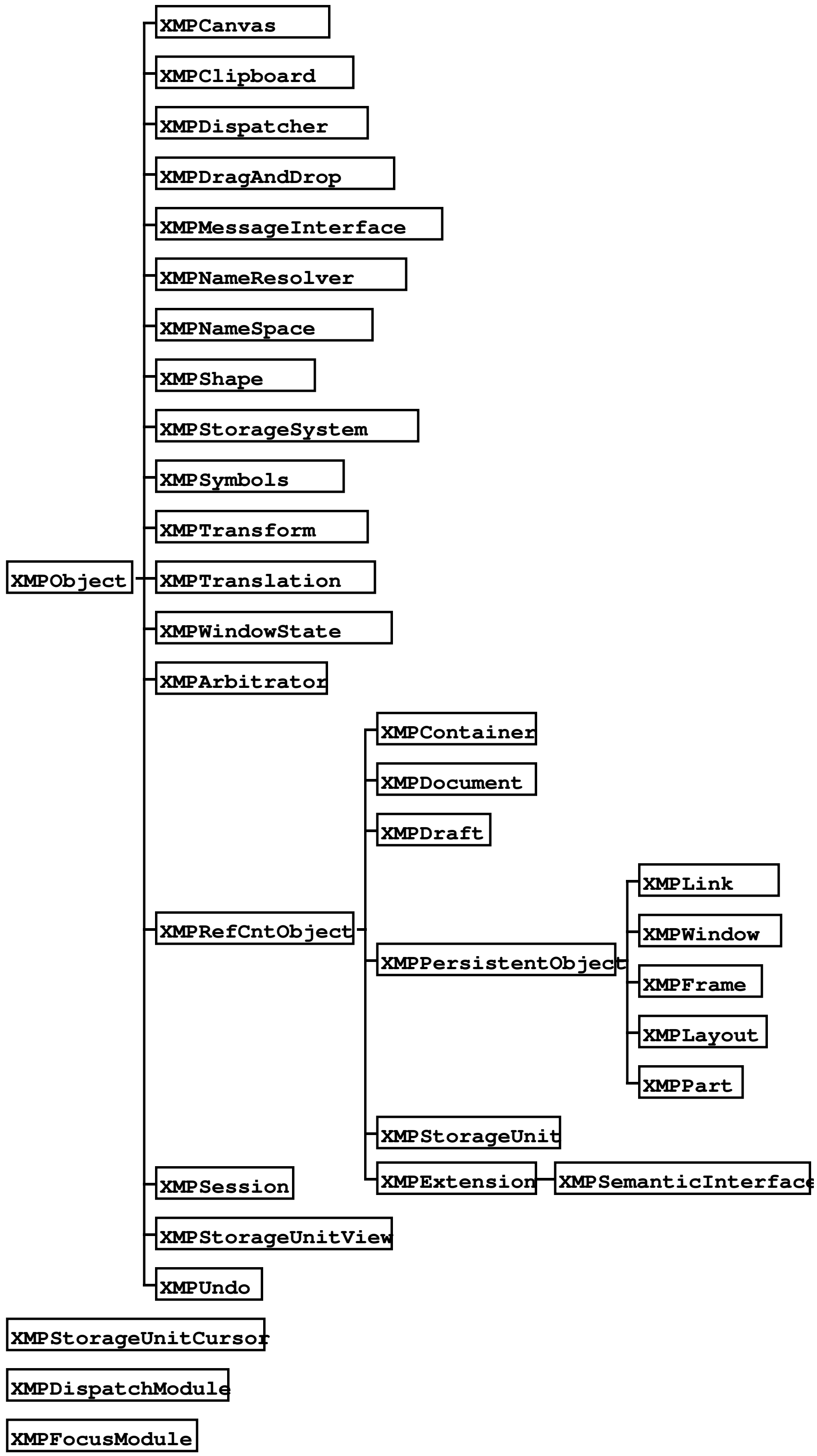


Figure 3.1 Abbreviated OpenDoc Class Hierarchy

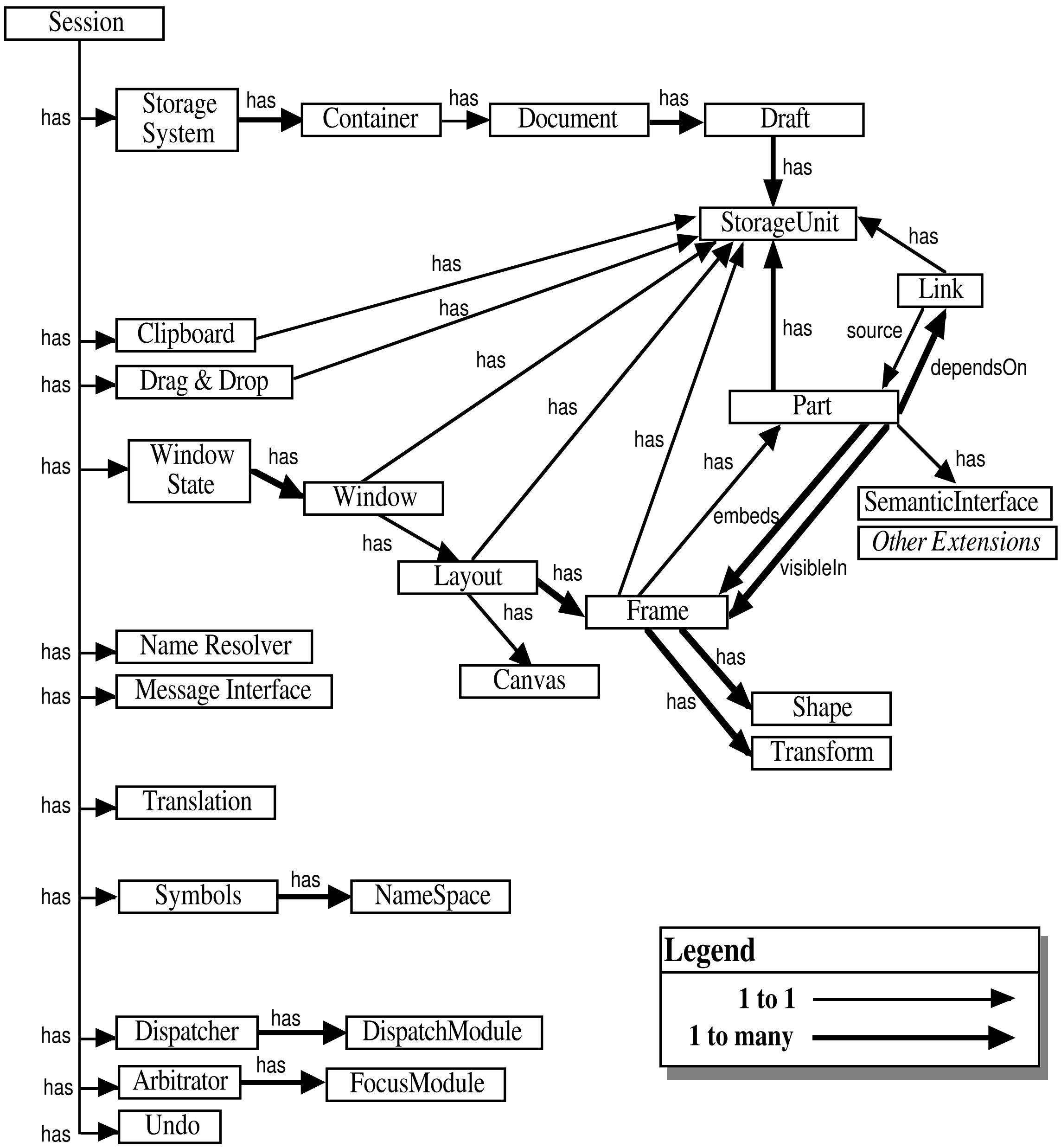


Figure 3.2 Abbreviated OpenDoc Object Model

CHAPTER 3. PROGRAMMER MODEL

This chapter is divided into four sections, each representing a particular kind of developer. They are:

Part handler (new)	These handlers represent new development efforts, from the ground up. Many part handlers will be created this way, but many will also be produced as refits of existing application code.
Part handler (refit)	These handlers are produced by refitting existing applications to match the OpenDoc model. They are easier to produce than writing from scratch, but may have limited functionality.
Container	These are applications modified to be containers of OpenDoc parts, thus replacing the standard document shell application.
Platform vendor	This is the effort needed to implement OpenDoc on a new platform.

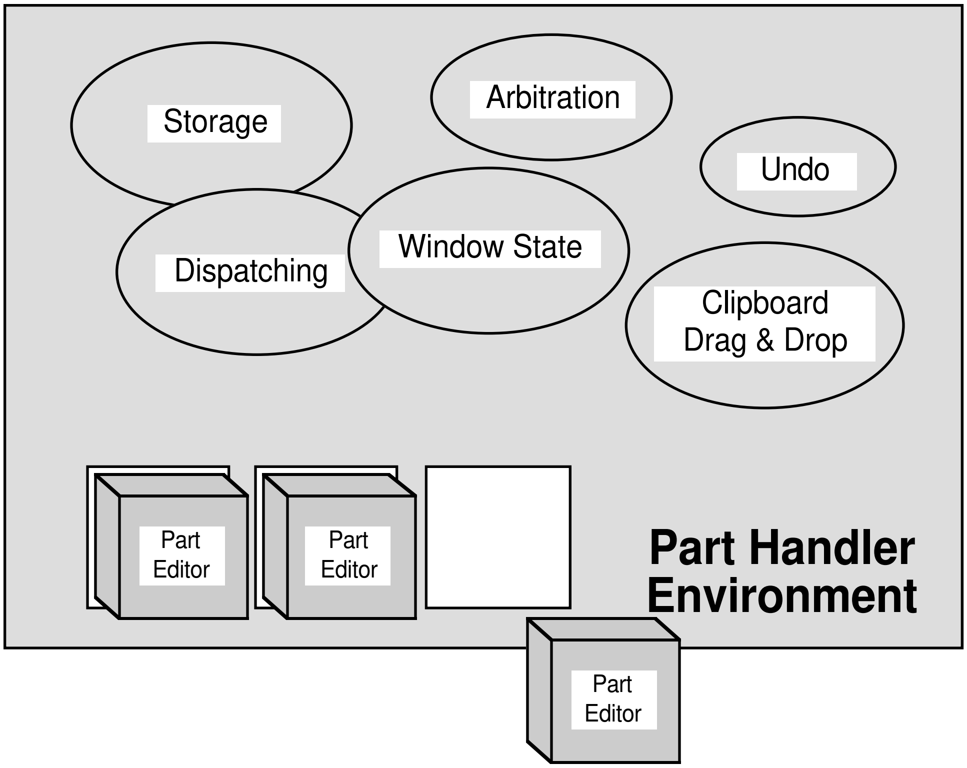
These sections tend to be a bit repetitive. It may make sense to read each of the sections independently, based on reader interest, rather than trying to wade through them in order. Many existing applications will be a combination of new implementation and refit, and so may straddle the various categories.

Programming environment

OpenDoc provides a new set of facilities which programmers may use to create new user functionality. These basic facilities allow a dynamically linked part handler to coexist with other handlers in the same document space.

This document space is called the OpenDoc document shell. Really, it is a special version of the traditional application. It provides a shared address space which the handlers can use to internalize document content. At the same time, it provides a set of global databases which handlers can use to track the behavior of the document and other handlers.

Here's an illustration of these global structures. This picture is not an exhaustive description, but gives a sense of the most important structures.



The various global structures are:

Storage This set of objects manages the persistent storage of the document, as well as providing memory management of the ephemeral storage required by the part handlers.

Dispatching This set of objects is responsible for delivering both semantic and UI events to the part handlers.

Arbitration These objects manage shared resources like the menubar, keystroke focus, etc.

Window state These objects manage the visible windows and their associated frames, so that handlers can cooperate to lay out graphics into the windows.

Undo These objects allow undo and redo to be coordinated in the document.

Clipboard, Drag & Drop These objects make it easier to transfer information between handlers.

Into this space are linked the part handlers which replace applications in OpenDoc. Each part handler is bound at run-time to the part information stored persistently in the document file.

Part Handler (new from scratch)

Produce content model

Developers must create a content model and construct semantic events which allow a full range of operations on the objects in their content model.

Implement core data engine

In any program which does useful work, there are a set of data structures which are manipulated according to some set of rules. It may or may not persistently store information, but in all cases data are manipulated according to some set of algorithms. We refer to this set of structures and rules, independent of any human interface, as the *core data engine* or *core engine* of a program.

While this is an obvious step, there is little to say from the OpenDoc point of view about how this is accomplished. It is clear that the engine should be able to free storage when requested, and should adequately handle cases where the part can only be partially read, handle any multiprocessing issues, etc.

A key element of this process, if script recordability is desired, is to make sure that the human interface interacts with the core engine through a restricted set of calls which match the user model of the core engine. These calls, which we term semantic events, are generally not useful as a high-bandwidth API to the core engine, but instead a user interface building toolkit. Given a bottleneck through which these semantic events pass, scripts can be recorded with robust, high quality semantic content.

Implement storage code

Basic storage of the part

To guarantee the collaborative aspects of OpenDoc it is necessary for the part handlers to utilize the Storage API and store the applicable standard properties with every part that they create or edit.

The part handler is responsible for implementing code to internalize and externalize the part. This need not mean a complete load into memory, nor a complete rewrite of the part content. However, the handler must be able to get the part into a workable state for accepting events and rendering requests, and it must be able to guarantee that the part has been satisfactorily written to the structured storage system. The part handler should never change the type of the part without an explicit user action. Any translations which must be done automatically will have been performed by the platform implementation.

The part handler implementor is also responsible for managing segmentation of his part handler, which is linked as a shared library. Lastly, he is responsible for managing global storage associated with the handler, as opposed to any particular part.

The developer view of the storage facility is quite different from the current file system and resource manager, but the change in viewpoint is necessary to enable the compound document features of OpenDoc.

Data transfer

Both the clipboard and drag & drop represent ways to transfer information between parts. OpenDoc supports the movement of compound data by providing the same storage abstraction for data transfer that is used for persistent storage. Thus, the clipboard and drag packages are represented as storage units, with an associated draft.

A slight addition is the notion of a promise, a special clipboard type which represents a promise of information which will later be forthcoming.

Linking

Linking support is really a combination of event handling code and storage code. Linking uses the same data transfer mechanisms mentioned above. However, XMPLink adds a set of notification calls which keep the destination of a link up to date. This mechanism can hide publish & subscribe as well as intra-document links.

Collaboration and drafts

Developers should not have to worry about drafts of OpenDoc documents normally. The document shell will handle the mundane aspects of drafts for the developer.

Implement rendering code

During rendering, the part handler is responsible for examining the frame and displaying the correct information, properly transformed and clipped, in the frame. The part handler is responsible for storing any needed information in the part info field of the frame.

The part handler should examine the frame to determine whether the frame is on a dynamic canvas or a static one, and perform frame negotiation accordingly. On a static canvas, the frame should attempt to add frames or extend them until all of its contents have become visible. On a dynamic one, assuming the part allows scrolling, it need only live with whatever frame the containing part offers.

If, during rendering, the part handler must get an embedded frame to render, it is responsible for adding that frame to the layout data structure.

If the embedded frame moves or is no longer visible, the embedding part handler is responsible for updating the layout or removing the frame from the layout.

If the part handler has more than one frame in which it is displayed, it is responsible for determining which of its frames are visible and updating those frames if an alteration is made in the part in response to some event. This may include invoking frame negotiations.

Implement event handling code

Having implemented a content model and semantic event handling, the developer must support direct manipulation of the part by implementing those routines of the part API which respond to UI events like mouse clicks and keystrokes. In addition to tracking the mouse and dealing with Drag-And-Drop, the part handler may take the appropriate steps to support scripting.

Part handlers must also implement the portions of the API which respond to activation and deactivation, by modifying the layout and configuration of the user interface. Part handlers must use the OpenDoc arbitrator API to manipulate the menu bar. If the remainder of a part's user interface can be contained within its frame, or in floating windows, existing code might port fairly easily. But part handlers might wish to display UI elements like rulers outside of the active frame, in which case they are getting involved in layout negotiation. Developers might also wish to switch to using OpenDoc UI parts rather than lower-level facilities.

There is a mechanism for extensions to the part API, so that the minimal API can be kept small. A part with a simple content model and a minimal direct manipulation interface should be easy to implement.

Implement scripting code

Part implementors who choose to support the semantic events extension must implement code to handle the semantic events defined in their content model.

Forward-thinking developers who already support AppleEvents and AppleScript will have done much of this work already.

Part handlers must provide accessor functions to resolve external references to a part's content objects. They must also provide semantic event handlers to implement their content operations. Part handlers must record which selections are exported as links, and notify dependent parts when linked content objects change.

By investing the effort in developing a content model, and implementing the routines for resolving object specifiers and handling semantic events, the developer of a part handler has ensured the separation of the part handler's "engine" from its user interface, and will reap great rewards. The part type is editable by all future interfaces which generate semantic events, including script, voice and pen interfaces. More mundane interfaces like menus and dialogs, built using state-of-the-art tools, can also invoke scripts or otherwise generate semantic events.

In order to allow script recording, the developer must generate and process semantic events as a result of any UI event, and process the semantic events through a standard bottleneck where a recorder may record the events. Note that this is also a good way to get ready to support new human interfaces like pen and voice recognition, and greatly enhances portability. Recording for its own sake may not be important enough to induce developers to follow a strict separation of UI and semantics, but the added bonus of better interfaces and portability probably makes this a good option.

To allow tinkering, a developer creates a scriptable and optionally recordable application. Before processing an event, the part handler must check to see if a script attached to the part wants to handle the event. If so, it must allow the script to run.

The part handler, if made scriptable, might activate UI part handlers for the menu bar, tool palettes and ribbons, and dialogs, rather than implementing these internally. Those UI part handlers invoke scripts that talk back to the original part and part handler that invoked them.

Implement desired extension interfaces

The part handler is responsible for implementing any desired extension APIs. These may cover any of a number of areas, and detailed descriptions are beyond the scope of this document. These could include full text search, spell checking, linking, or many other possible interactions. In general, these APIs are reserved for those areas whose bandwidth or integration requirements deny the use of scripting to accomplish the interaction.

The CIL will be actively involved in proposing and publishing standard interfaces between parts.

Package the part handler

The developer, as part of packaging the part for shipment, should attach information indicating what part types are handled, what semantic events are handled, and what extension APIs are handled.

Create stationery

Once a working part handler is available, the developer will need to produce some stationery which has an empty copy of his part type. This stationery will be used by end users to insert new parts of the correct type.

Additional steps if writing a containing part handler

Add embedded parts to content model

A named object type of the content model of any containing part, where embedded parts can be added, must be defined.

Add frame management to event handling code

Once embedded parts are present, any event handling code which changes a frame will need to include code to notify the part in the frame of the change, and code to maintain the shape and transformations of the frame itself.

If, during rendering, the part handler must get an embedded frame to render, it is responsible for adding that frame to the layout data structure, and asking the frame to render itself.

If the embedded frame moves or is no longer visible, the embedding part handler is responsible for updating the layout or removing the frame from the layout.

Add frame management to rendering code

Rendering code in a containing part must include layout negotiation support, and must update the transformations associated with each visible embedded frame.

Add embedding support to storage code

For the containing part, this is relatively simple. If it wishes to write out an embedded part, it can simply ask that the storage unit be copied to the new location, and the OpenDoc libraries will take care of the data copying.

Part Handler (refit existing application)

Determine content model

Existing applications, which do not have a defined content model, may have to retroactively determine one. This process is necessary only if linking or semantic events are to be supported by the part handler.

In general, this process is one of examining application behavior and abstractions to come up with a content model.

Factor core data engine

Again, if scripting is to be supported, particularly tinkerability and recordability, the application code must be factored into a core data processing engine, driven by semantic events, and a presentation driven by HI events. Once this factoring has taken place, the application is ready to have scripting code fitted onto it.

Refit storage code

Because OpenDoc part handlers must share persistent storage, it is necessary to refit any file I/O or clipboard I/O into OpenDoc terms. This is generally very simple. The Handler need only know the correct property name and have a defined value type to read and write into an OpenDoc value stream. The remaining code, assuming it is stream oriented as most applications are, can simply be used with the OpenDoc read and write calls.

Refit rendering code

Because OpenDoc layout is shared, a part must refit its rendering code to handle existence in an OpenDoc frame. Again, this is straightforward if the code already uses the platform window clipping tools.

Beyond this simple clipping change, the part must also make decisions about how it wants to respond to changes in its enclosing frame. The simplest decision is to simply clip to whatever size is given. However, scaling to fit, attempting to resize or asking for added frames are also options, to be decided based on user requirements.

Refit event handling code

The event handling refit is probably the most serious step besides scripting support. The developer must change his code to handle the menu bars, palettes, and dialogs to match the OpenDoc model. In general, this involves testing for activity (whether the part already has the needed foci) and grabbing menu, keystroke, and selection focus as needed.

Once these are obtained, the part is free to put up dialog boxes, change the menu bar, or put up palettes as needed. When focus is lost, the part handler should remember the fact and hide windows or palettes, and also get ready to reacquire foci if needed later.

Refit scripting code

Once the basics of scripting: content model and factoring have been defined, the rest of it is relatively easy. Installing event handlers and object accessors is very similar to the process for doing so in a normal application. Assuming that the applications is already using the AppleEvents and the open scripting architecture, most of these will have already been completed by the developer. In OpenDoc, the SemanticInterface object is the location for installation, instead of using the normal Apple Event Manager call. For sending events, the MessageInterface is used instead of the AE Manager.

Container

It will be very useful to refit large existing applications to be part containers. These documents will still be owned by their application, but that application will be reworked to add OpenDoc parts embedded in their content.

This task is not trivial, but is in fact relatively simple. It is more difficult than the QuickTime revision, but can probably be accomplished with a week or two of programmer time.

Add some initialization code

A container only application will need to call the same initialization call to the OpenDoc libraries that the document shell application would normally call.

Refit storage code

Because OpenDoc part handlers must share persistent storage, it is necessary to refit any file I/O or clipboard I/O into OpenDoc terms. This is generally very simple. The Handler need only know the correct property name and have a defined value type to read and write into an OpenDoc value stream. The remaining code, assuming it is stream oriented as most applications are, can simply be used with the OpenDoc read and write calls.

Thus, the application must still be willing to use the OpenDoc file format for files that contain parts. The application will be given a special property at the document level which it can use for stream oriented file I/O. We will provide a small library with an appropriate routine to return an XMPStorageUnitView which the container can use for writes and reads.

All embedded parts can then use the ordinary storage system calls, without interfering with the document.

Refit event handling code

The document will then need to handle events by passing events to the OpenDoc dispatcher first, and handling them only if OpenDoc does not.

To accomplish this, we will define a simple "do nothing" part handler for the root frame of every container window. We'll provide a simple library call to create a window with this part handler to work with OpenDoc. This "do nothing" handler will grab mouse, selection, and menu foci (possibly others) when mouse clicks land in the container's space, and will then simply respond that the event was not handled, giving the containing application a chance to do it.

Add embedded parts to content model

A named object type of the content model of the container, where embedded parts can be added, must be defined.

Add frame management to event handling code

Once embedded parts are present, any event handling code which changes a frame will need to include code to notify the part in the frame of the change, and code to maintain the shape and transformations of the frame itself.

If, during rendering, the container must get an embedded frame to render, it is responsible for adding that frame to the layout data structure, and asking the frame to render itself.

If the embedded frame moves or is no longer visible, the container is responsible for updating the layout or removing the frame from the layout.

Add frame management to rendering code

Rendering code in a containing part must include layout negotiation support, and must update the transformations associated with each visible embedded frame. This includes printing code.

Add embedding support to storage code

For the container, this is relatively simple. If it wishes to write out an embedded part, it can simply ask that the storage unit be copied to the new location, and the OpenDoc libraries will take care of the data copying.

This means that the container should probably use OpenDoc calls to write to the clipboard or drag & drop.

Platform Vendor

Implement a storage system

The platform vendor is responsible for implementing the storage protocol between part handlers and the document shell. Since the document shell is the true owner of the document file, it will need to implement all of the necessary code to apply OpenDoc policies atop the structured storage facility of the platform for which the document shell is being written.

Classes XMPStorage, XMPContainer, XMPDocument, XMPDraft, XMPClipboard, XMPStorageUnit, XMPStorageUnitCursor, XMPStorageUnitView, XMPLink and XMPDragAndDrop, along with their attendant iterators must be implemented. It is quite possible, even likely, that more than one implementation of XMPContainer, XMPDocument, XMPDraft, and XMPStorageUnit will exist on a platform.

See the header files and class documentation in the *OpenDoc API Description* for detailed specifications.

Implement an arbitrator

The platform vendor must implement any HI or OS policies needed to make a complete arbitrator instantiation.

Class XMPArbitrator, along with its auxiliary classes and iterators, will need to be implemented.

See the header files and class documentation *OpenDoc API Description* for detailed specifications.

Implement event delivery

The platform implementor implements the delivery of events to the individual part handlers. It must accept both semantic events and UI events and deliver them correctly to the part handlers.

Classes XMPDispatcher, XMPUndo, XMPMessageInterface, XMPNameResolver, XMPSemanticEvents and XMPSemanticInterface, along with their associated iterators and so on must be created.

See the header files and class documentation *OpenDoc API Description* for detailed specifications.

Implement a window & layout system

The platform implementor implements the layout structures used by parts during layout for screen or printing. OpenDoc relies on platform facilities to perform drawing, provide coordinate spaces for layout, and all other graphical tasks. The platform implementor may need to alter the implementation of any of the window state and layout classes to suit a local environment.

Classes XMPWindowState, XMPLayout, XMPWindow, XMPFrame, XMPCanvas, XMPTransform and their attendant iterators and such must be implemented. If two or more drawing systems reside on the platform, implementation of these classes may involve transforming shapes and points between coordinate spaces.

See the header files and class documentation *OpenDoc API Description* for detailed specifications.

Implement a runtime environment

A runtime environment must be created, with binding support, support for finding global document structures, exception handling, translations, and memory management. These structures may vary significantly from one platform to another.

A basic set of them, defined in XMPSession, XMPDraft, XMPNameSpace and XMPTranslation appear likely to be common across a variety of platforms.

See the header files and class documentation *OpenDoc API Description* for detailed specifications.

CHAPTER 4. OPENDOC API

Notes about Style

Language Choice

The OpenDoc APIs have been constructed in C++, a language chosen for its object dispatching capabilities as well as its relative portability. In later releases, these definitions will be written in a CORBA-compliant interface definition language, to support language-neutral distributed dispatching.

Distributed operation

In the interim, though, several steps have been taken to support distributed operation.

First, all major objects are instantiated by factory objects, rather than constructors. This does not apply to all objects, but generally applies to every object likely to be used remotely.

Secondly, we have attempted to minimize the frequency of inter-part calls. In some cases, like layout, frequent calls cannot be avoided. However, in general the number of inter-part and library calls has been kept as small as possible.

PlfmType.h

This file indicates the list of platform specific types used by OpenDoc's platform implementations and platform facilities. It is used to segregate these types from those defined to support common implementations. Each of these types should be redefined on each platform.

XMPTypes.h, XMPTypes.r

These files include types which are used in common implementations as well as platform implementations. In general, types in these files should not be changed across platforms. Note that many types defined here depend on platform specific types, which should be changed for each platform.

Exception handling

OpenDoc includes an exception handling system similar to the one proposed for C++. However, the OpenDoc implementation does not use the storage management or multiple signal types capability of the proposed standard. This allows us to produce C bindings for the OpenDoc Part API. In the future, this will be altered to match the CORBA standard for exception handling.

Class information

Of primary interest to the reviewer will be the theory of operation section and the comments attached to each member of the class definitions; please refer to the *OpenDoc API Description*.

In certain cases, the APIs are Macintosh specific, because we are evaluating their workability first on Macintosh, the platform most familiar to the team. We are especially interested in hearing about places where we have made the interfaces too Macintosh specific.

Class categories

For the purposes of this document, we've divided the classes up into four major groupings, each of which we'll explain in greater detail later. These include:

Developer implementations: These classes must be supplied by developers. They are typically part handlers which manage some sort of useful content, such as text editors, graphics packages, etc.

Platform facilities: simple cover classes which provide handy pointers to basic platform systems like the drawing engine.

Platform implementations: Classes which are highly dependent on platform specifics, but which should provide extremely similar APIs across platforms.

Common implementations: These are classes which are almost independent of platform specifics in their implementation. They may be replaced by any platform implementor, but we believe extensive changes are unlikely.

Developer implementations

Developers implement these classes. In fact, the vast majority of an OpenDoc document's behavior is encapsulated in these classes.

Part

This class is the basic set of entry points which must be defined for any part editor or viewer. This is the major learning curve for developers.

Each developer will produce a class derived from part. We will provide as much basic implementation as we can to allow a part to behave properly, but in general the developer will have about 50 calls to implement. Of these, most are related to event handling, with relatively few related to storage issues and layout negotiation.

Developers who implement a part which can embed others must extend the behavior of part to include negotiation about the position and shape of embedded frames, and about their visibility.

Dispatch Module

In certain special cases, a developer may wish to gain access to low level operating system events which are not covered by the OpenDoc API. In these cases, the developer will probably wish to define a new focus in the arbitrator, and deliver this class of events to the owner of that focus. To get these events delivered, the developer will need to extend the dispatcher using a dispatch module. Such modules may need to examine the arbitrator or window state to decide which part should receive the event.

Platform facilities

These classes encapsulate extremely platform specific implementations. These classes are so completely specific that the class definitions are simply cover APIs, with no significant level of implementation in the reference release.

Canvas

This class represents a black-box encapsulator of the basic platform's drawing environment. This represents a grafport on a Macintosh using QuickDraw Classic, but might be any of several other drawing context structures on other platforms.

Shape

This class is a black-box encapsulation of the basic clipping shape structure. On Mac or Windows, this might be a region. On other systems, it might be a path or even more sophisticated structure. On less capable systems, these might be simple rectangles or rectangle sets.

Transformation

A black-box encapsulation of the basic viewing transformations available on the platform. These transformations must be composable and invertable, but we use no other operations in OpenDoc.

Platform implementations

These classes must be developed by platform vendors. They are in general quite platform-specific. The platform vendor must produce an implementation which closely matches the reference release in semantics, using platform specific facilities.

Iterators

OpenDoc defines a number of iterator classes. These are fairly standard members of the breed. On some platforms, these implementations will need to be process/thread safe, but on others they will need no such protection. It is up to the platform implementors to make sure these iterators and the collections they iterate over are appropriate for the platform.

Arbitrator

The arbitrator is used to allow parts to negotiate about shared resources. Some examples of these might include the keystroke focus, the menu bar, serial ports, or any of a wide array of other resources. Each of these resources is termed a focus, and different platforms will support differing foci.

Another area in which platform specific work will need to be done is in implementing the specific behavior associated with various foci. On the Macintosh, for instance, the serial ports are shared across processes, so the arbitrator will need to globally manage serial port access.

Dispatcher

This class is used to correctly distribute user interface and other low-level operating system events to the parts of the document.

The dispatcher is different on each platform, because it encapsulates many platform-specific HI rules. Platforms vary widely with respect to keyboard events, for instance. Some associate keystroke delivery with cursor position, some with a insertion focus. The dispatcher must implement all of the platform's basic HI rules when determining where events are to be sent.

If needed, the dispatcher can be extended using dispatch modules, mentioned below as a developer implementation. However, a platform implementor will probably also use the dispatch module to extend the dispatcher as the platform evolves.

Storage

The storage object manages access to all of the containers used by a document process. Since this class is responsible for interpreting file requests on the platform, it will almost certainly be re-implemented on each platform.

Container

Containers generally represent some form of physical storage containment. However, in a sophisticated object store, the notion of physical containment may be merely illusion. In either case, the implementation of container must reflect underlying platform implementations.

Document

A document represents the user concept of a document, a handy container of content. Documents serve primarily as a manager of revisions and drafts.

There may be several implementations of document, depending on the kind of physical container in which the document resides.

Draft

Drafts store all of the first-class persistent objects of a particular saved state in a document. A draft consists of a set of storage units, which represent both the meta-data and the actual content of a document.

There may be several implementations of draft, depending on the kind of physical container in which the draft resides.

Storage Unit

These represent the fundamental unit of persistent identity in a document. They consist of a list of properties, each of which is a list of values. Only storage units have persistent identity.

There may be several implementations of storage unit, depending on the kind of physical container in which the storage unit resides.

Clipboard

A clipboard is a data transfer structure designed to buffer data temporarily. This gives a measure of time independence to the data transfer. This same technique can also be used to give location independence to such transfers, as in the case of a network clipboard. In OpenDoc, each data transfer is conducted using a storage unit, to provide full compound content transfer capability.

Storage Unit

We expect the storage unit used in a clipboard to have special, platform specific behavior. On platforms with existing clipboard mechanisms, human interface standards about what types should appear on the clipboard may exist. In these cases, the storage unit should recognize these types when they are written to the storage unit, and redundantly store them to the platform clipboard in the approved fashion.

Implicit in the existence of a storage unit is the existence of a containing draft. It is in this draft that any embedded parts and frames would be transferred along with the basic data being transferred. This probably implies implementing a specialized draft class to hold this information in memory or in a specified temporary file.

Drag & Drop

Drag and Drop class represents a transfer of data using direct manipulation. It is quite similar to a clipboard transfer, but is more ephemeral. In OpenDoc, each data transfer is conducted using a storage unit, to provide full compound transfer capability.

Storage Unit

We expect the storage unit used in a drag & drop transaction to have special, platform specific behavior. On platforms with existing drag & drop mechanisms, human interface standards about what types should appear in the transaction may exist. In these cases, the storage unit should recognize these types when they are written to the storage unit, and redundantly store them to the platform transaction structure in the approved fashion.

Link

Links represent persistent data transfer conduits. As with other OpenDoc transfer mechanisms, the data are passed using a storage unit.

Storage Unit

We expect the storage unit used in a persistent link to have special, platform specific behavior. On platforms with existing linking mechanisms, human interface standards about what types should appear in the link may exist. In these cases, the storage unit should recognize these types when they are written to the storage unit, and redundantly store them to the platform link structure in the approved fashion.

Semantic Interface

The semantic interface is invoked by either the document shell or the dispatcher, depending on the platform. The platform may have specific requirements around event dispatching as well, and may need to alter this class to perform the correct behavior during handler invocation. This is likely to occur if parts are accessed using interprocess calls.

Message Interface

MessageInterface encapsulates the sending and reception of semantic events. As such, it is extremely likely to change from platform to platform.

Common implementations

These classes are implemented by the platform vendor, just as the previous section. However, we believe these classes are so generic as to obviate the need for extensive change by a platform developer. In effect, these are provided in working form by the reference implementation.

Name Resolver

Semantic events will wish to name content objects within a part. the name resolver helps parts resolve these names. Because the process of resolving names is so generic, and is specialized by developer produced callbacks, this class is unlikely to need reimplementation unless the memory management scheme is changed.

Name Space

Name spaces are a generalized binding mechanism allowing association of some kind of data with a name. We plan to use this class in implementing run-time binding of part types to part handler implementation. This is a very generic process, one unlikely to need change.

Window State

The window state manages the document shell's global list of visible windows. This is primarily used by the dispatcher, but may also be used by developers in creating part handlers which interact with the window system. This generic behavior is unlikely to change radically from system to system.

Window

A window keeps the organization of visible frames associated with a platform window. This class keeps a black-box reference to the actual platform window. Since the window object mainly keeps lists of z-ordered frames, it is an extremely generic implementation.

Frame

Frame keeps black-box references to a number of shape objects. It also provides a notification service to both the container and embedded parts associated with the frame. Again, this implementation is extremely generic. The shape and part implementations provide much of the intelligence associated with layout .

Future areas for expansion

Several major areas have been planned for but not implemented in the OpenDoc architecture. Because of time and platform constraints, we have chosen not to include these features in the first OpenDoc release.

Authorization

Given operating system support, we envision using the semantic interface as a bottleneck for access control and authorization. However, insufficient support exists across the desired platforms for storing and editing access information. Due to time constraints, we have chosen not to attempt to implement a common solution.

Store & forward messaging

Eventually, we hope to use the same semantic interface to deliver store & forward messages as well as real-time connections. This raises a series of interface issues which have not been exhaustively analyzed at this time.

Digital signature & encryption

Digital signatures of compound documents with shared data presents interface and storage issues. We believe we understand them at this time, but are not willing to implement them in this release due to time constraints and lack of consistent platform support.

Compound Human Interface

The prospect of easily composed human interfaces built using parts and compound document techniques is presently handled by the architecture, but there are as yet unresolved issues about runtime support and the human interface for customizing the compound human interface.

Detailed APIs

A summary of the APIs of most interest to developers is provided in Appendix A. See the *OpenDoc API Description* for detailed documentation and header files.

Appendix A. Public API Subset

Here is a summary of the APIs most interesting to developers. This subset is not complete, and has several following sections which specify extended areas of functionality not generally of interest to developers. Calls which are italicized are Macintosh specific, and would not be a part of a reference release.

Basic part development

These are the calls which a part handler developer must implement. They represent the basic development interface in OpenDoc.

XMPPart

```
void Draw(
        XMPFrame* frame,
        XMPShape* invalidShape)
XMPBoolean HandleEvent(
        XMPFrame* frame,
        XMPEventData event)
void Internalize()
void Externalize()
void Open(XMPFrame* frame)
void UndoAction(XMPActionData actionState)
void RedoAction(XMPActionData actionState)
void DragEnter(
        XMPStorageUnit* dragInfo,
        XMPFrame* frame,
        XMPPoint where)
void DragLeave(
        XMPFrame* frame,
        XMPPoint where)
void DragWithin(
        XMPStorageUnit* dragInfo,
        XMPFrame* frame,
        XMPPoint where)
XMPDropResult Drop(
        XMPStorageUnit* dropInfo,
        XMPFrame* frame,
        XMPPoint where)
void DropCompleted(
        XMPPart* destPart,
        XMPDropResult dropResult)
void FulfillPromise(XMPPtr promiseData)
XMPLink* CreateLink(
        XMPObjectSpec* selectedContent)
void LinkUpdated(XMPLink* updatedLink)
XMPBoolean BeginRelinquishFocus(
        XMPTypeToken focus,
        XMPFrame* ownerFrame,
        XMPFrame* proposedFrame)
void AbortRelinquishFocus(
        XMPTypeToken focus,
        XMPFrame* ownerFrame,
        XMPFrame* proposedFrame)
void CommitRelinquishFocus(
        XMPTypeToken focus,
```

```
        XMPFrame* ownerFrame,
        XMPFrame* proposedFrame)
void FocusAcquired(
        XMPTypeToken focus,
        XMPFrame* ownerFrame)
void FocusLost(
        XMPTypeToken focus,
        XMPFrame* ownerFrame)
void FrameShapeChanged(XMPFrame* frame)
XMPShape* RequestFrameShape(
        XMPFrame* frame,
        XMPShape* frameShape)
void AddDisplayFrame(
        XMPFrame* frame,
        XMPName* viewType)
void RemoveDisplayFrame(XMPFrame* frame)
XMPFrame* CreateEmbeddedFrame(
        XMPFrame* containingFrame,
        XMPShape* frameShape,
        XMPTransform* externalTransform,
        XMPPart* embedPart,
        XMPID frameGroupID,
        XMPBoolean isOverlaid)
void RemoveEmbeddedFrame(XMPFrame* frame)
void ChangePresentation(
        XMPFrame* frame,
        XMPName* viewType)
XMPStorageUnit* GetContainingPartProperties(
        XMPFrame* displayFrame)
void ContainingPartPropertiesChanged(
        XMPFrame* displayFrame,
        XMPStorageUnit* propertyUnit)
XMPPtr ReadActionState(
        XMPStorageUnitView* storageUnitView)
void WriteActionState(
        XMPPtr actionState,
        XMPStorageUnitView* storageUnitView)
void DisposeActionState(
        XMPActionData actionState,
        XMPDoneState doneState)
XMPPtr ReadPartInfo(
        XMPStorageUnitView* storageUnitView)
void WritePartInfo(
        XMPPtr partInfo,
        XMPStorageUnitView* storageUnitView)
XMPPart(
        XMPStorageUnit* storageUnit,
        XMPSession* session);
~XMPPart()
XMPID GetID()
XMPStorageUnit* GetStorageUnit()
XMPSize Purge(XMPSize size)
XMPBoolean HasExtension(
        XMPType extensionName);
XMPExtension* GetExtension(
        XMPType extensionName);
```



```
void  ReleaseExtension(
       XMPExtension* extension);
```

void MouseEnterFrame(XMPFrame frame, XMPPoint where)*

void MouseLeaveFrame(XMPFrame frame, XMPPoint where)*

XMPExtension

```
XMPObject* GetBase();
void Release()
```

Basic part environment

These are the basic calls which could be of interest to the part developer. This initial section represents those calls the developer of a simple part handler would wish to use. Some viewers, which have no editing behavior and do not allow changes to their contents, could use a much smaller subset of this API.

XMPArbitrator

```
XMPBoolean RequestFocusSet(
       XMPFocusSet* focusSet,
       XMPFrame* requestingFrame);
void RelinquishFocusSet(
       XMPFocusSet* focusSet,
       XMPFrame* relinquishingFrame);
```

XMPCanvas

```
XMPCanvas(
       XMPGraphicsSystem graphicsSystem,  XMPPlatformCanvas
       platformCanvas,
       XMPBoolean  isDynamic,
       XMPBoolean  isOffscreen);
~XMPCanvas();
XMPGraphicsSystem GetGraphicsSystem();
XMPPlatformCanvas GetPlatformCanvas();
XMPBoolean IsDynamic();
XMPBoolean IsOffscreen();
```

XMPClipboard

```
void Clear();
XMPStorageUnit* GetStorageUnit();
```

XMPDispatcher

void SetMouseRegion(XMPRgnHandle area);

XMPRgnHandle GetMouseRegion();

void RegisterIdleFrame(
XMPFrame frame,*
XMPIdleFrequency frequency);

void UnregisterIdleFrame(XMPFrame frame);*

XMPDraft

```
XMPDraftID GetID();
XMPDraftName GetName();
void SetName(XMPDraftName name);
void Release();
XMPStorageUnit* CreateStorageUnit();
XMPStorageUnit* GetStorageUnit(
       XMPStorageUnitID id);
```

```
void RemoveStorageUnit(
       XMPStorageUnit* storageUnit);
void RemoveFromDocument();
XMPBoolean ChangedFromPrev();
XMPFrame* CreateFrame(
       XMPFrame* containingFrame,
       XMPShape*  frameShape,
       XMPPart*  part,
       XMPUShort  frameGroup,
       XMPBoolean  isOverlaid);
XMPFrame* GetFrame(XMPStorageUnitID id);
void RemoveFrame(XMPFrame* frame);
XMPPart* CreatePart(XMPType partType);
XMPPart* GetPart(XMPStorageUnitID id);
void RemovePart(XMPPart* part);
XMPLink* CreateLink();
XMPLink* GetLink(
       XMPStorageUnitID id,
       XMPLinkSpec* theLinkSpec);
void RemoveLink(XMPLink* link);
XMPWindow* CreateWindow(
       XMPPlatformWindow newWindow,
       XMPFrame* sourceFrame);
XMPWindow* GetWindow(XMPStorageUnitID id);
XMPLayout* CreateLayout(
       XMPFrame* containingFrame,
       XMPShape* frameShape,
       XMPPart* part,
       XMPUShort frameGroup,
XMPLayout* GetLayout(XMPStorageUnitID id);
void  RemoveLayout(XMPLayout* layout);
XMPDraftPermissions GetPermissions();
```

XMPDragAndDrop

```
void Clear();
XMPStorageUnit* GetStorageUnit();
XMPDropResult StartDrag(
       XMPFrame* srcFrame,
       XMPValueType imageType,
       XMPPtr image,
       XMPPart** destPart,
       XMPPtr refCon);
```

XMPFocusSet

```
XMPFocusSet();
~XMPFocusSet();
void Add(XMPTypeToken focus);
```

XMPFrame

```
XMPFrame*     GetContainingFrame();
void SetContainingFrame(XMPFrame* frame);
XMPLayout* GetLayout();
void   SetLayout(XMPLayout* layout);
XMPWindow* GetWindow();
XMPCanvas* GetCanvas();
```

```
XMPULong GetFrameGroup();
void    SetFrameGroup(XMPULong groupID);
XMPPtr GetAttributeBundle();
void SetAttributeBundle(
        XMPPtr attributeBundle);
XMPBoolean IsOverlaid();
void    SetOverlaid(XMPBoolean isOverlaid);
XMPBoolean IsFrozen();
void    SetFrozen(XMPBoolean isFrozen);
XMPBoolean UsesCachedImage();
void    SetUsesCachedImage(
        XMPBoolean usesCachedImage);
XMPBoolean DoesPropagateEvents();
void    SetPropagateEvents(
        XMPBoolean doesPropagateEvents);
XMPPart* GetPart();
void ChangePart(XMPPart* part);
XMPInfoType GetPartInfo();
void    SetPartInfo(XMPInfoType partInfo);
XMPShape* GetFrameShape();
void ChangeFrameShape(XMPShape* shape);
XMPShape* RequestFrameShape(XMPShape* shape);
XMPShape* GetClipShape();
void ResetClipShape();
void IntersectClipShape(XMPShape* sectShape);
void SubtractClipShape(XMPShape* diffShape);
XMPShape* GetAggregateClipShape();
XMPShape* GetUsedShape();
void ChangeUsedShape(XMPShape* shape);
XMPShape* GetActiveShape();
void ChangeActiveShape(XMPShape* shape);
XMPTransform* GetExternalTransform();
XMPTransform* GetInternalTransform();
void ChangeExternalTransform(
        XMPTransform* transform);
void ChangeInternalTransform(
        XMPTransform* transform);
XMPTransform* GetAggregateExternalTransform();
XMPTransform* GetAggregateInternalTransform();
XMPBoolean IsDroppable();
void    SetDroppable(XMPBoolean isDroppable);
XMPBoolean IsDragging();
void    SetDragging(XMPBoolean isDragging);
void Invalidate(XMPShape* invalidShape);
void InvalidateAggregateTransform();
void Draw(XMPShape* invalidShape);
XMPHighlight GetHighlight();
void ChangeHighlight(XMPHighlight highlight);
void Release();
```

XMPLayout

```
XMPCanvas* GetCanvas();
void SetCanvas(XMPCanvas* canvas);
XMPWindow* GetWindow();
void SetWindow(XMPWindow* window);
void AddFrame(
       XMPFrame* frame,
       XMPFrame* siblingFrame,
       XMPFramePosition position);
void RemoveFrame(XMPFrame* frame);
void Internalize();
void Externalize();
```

XMPLink

```
XMPStorageUnit* GetStorageUnit();
XMPID GetID();
void ContentChanged();
void RegisterDependent(XMPPart* clientPart);
void UnregisterDependent(XMPPart* clientPart);
void Externalize();
```

XMPMenuBar

```
void AddMenuBefore(
       XMPMenuID menuID,
       XMPPlatformMenu menu,
       XMPPart* part,
       XMPMenuID beforeID);
void AddMenuLast(
       XMPMenuID menuID,
       XMPPlatformMenu menu,
       XMPPart* part);
void Display();
```

`void RegisterCommand(`
`       XMPCommandID command,`
`       XMPMenuID menu,`
`       XMPMenuItemID menuItem);`

`XMPBoolean IsCommandRegistered(`
`       XMPCommandID command);`

`XMPBoolean IsCommandSynthetic(`
`       XMPCommandID command);`

XMPNameSpace

```
XMPISOStr GetName();
void Register(XMPISOStr key, XMPPtr value);
void Register(XMPOSType key, XMPPtr value);
void Register(XMPSLong key, XMPPtr value);
void Unregister(XMPISOStr key);
void Unregister(XMPOSType key);
void Unregister(XMPSLong key);
XMPPtr GetValue(XMPISOStr key);
XMPPtr GetValue(XMPOSType key);
XMPPtr GetValue(XMPSLong key);
XMPValueIterator* CreateValueIterator();
```

XMPShape

```
XMPShape(
       XMPGraphicsSystem graphicsSystem,
       XMPPlatformShape platformShape);
~XMPShape();
XMPGraphicsSystem GetGraphicsSystem();
XMPPlatformShape GetPlatformShape();
void SetPlatformShape(
       XMPGraphicsSystem graphicsSystem,  XMPPlatformShape
       platformShape);
void CopyFrom(XMPShape* sourceShape);
XMPBoolean IsSameAs(XMPShape* compareShape);
void Subtract(XMPShape* diffShape);
void Intersect(XMPShape* sectShape);
XMPBoolean ContainsPoint(XMPPoint point);
void Transform(XMPTransform* transform);
```

XMPStorageSystem

```
XMPContainer* CreateContainer(
       XMPContainerType containerType,
       XMPContainerID id);
XMPContainer* GetContainer(
       XMPContainerType containerType,
       XMPContainerID id);
void NeedSpace(
       XMPSize memSize,
       XMPBoolean doPurge);
```

XMPStorageUnit

```
XMPDraft* GetDraft();
void Release();
XMPStorageUnit* Focus(
       XMPPropertyName propertyName,
       XMPPositionCode propertyPosCode,
       XMPValueType valueType,
       XMPValueIndex valueIndex,
       XMPPositionCode valuePosCode);
XMPStorageUnit* Externalize();
XMPStorageUnit* Internalize();
XMPID  GetID();
XMPStorageUnitName GetName();
void SetName(XMPStorageUnitName name);
XMPStorageUnit* AddProperty(
       XMPPropertyName propertyName);
XMPStorageUnit* AddValue(XMPValueType type);
XMPStorageUnit* Remove();
void  CopyFrom(XMPStorageUnit* fromSU);
XMPValueType GetType();
void  SetType(XMPValueType valueType);
XMPULong  GetValue(
       XMPULong length,
       XMPValue value);
void  SetValue(
       XMPULong length,
       XMPValue value);
```

```
void  InsertValue(
       XMPULong length,
       XMPValue value);
void  DeleteValue(XMPULong length);
```

XMPStorageUnitCursor

```
XMPStorageUnitCursor(
       XMPPropertyID propertyID,
       XMPValueType valueType,
       XMPValueIndex valueIndex);
XMPStorageUnitCursor(
       XMPStorageUnitCursor old,
       XMPStorageUnit* su,
       XMPPositionCode propertyPosCode,
       XMPPositionCode valuePosCode);
~XMPStorageUnitCursor();
```

XMPStorageUnitView

```
XMPStorageUnitView* Externalize();
XMPStorageUnitView* Internalize();
XMPStorageUnit* GetStorageUnit();
XMPStorageUnitName GetName();
void SetName(XMPStorageUnitName name);
XMPStorageUnitView* AddProperty(
       XMPPropertyName propertyName);
XMPStorageUnitView* AddValue(
       XMPValueType type);
XMPStorageUnitView* Remove();
void CopyFrom(XMPStorageUnit* fromSU);
XMPValueType GetType();
void SetType(XMPValueType valueType);
XMPULong  GetValue(
       XMPULong length,
       XMPValue value);
void  SetValue(
       XMPULong length,
       XMPValue value);
void  InsertValue(
       XMPULong length,
       XMPValue value);
void  DeleteValue(XMPULong length);
```

XMPSymbols

```
XMPNameSpace* CreateNameSpace(
       XMPISOStr  spaceName,
       XMPNSTypeSpec contentType,
       XMPNameSpace* inheritsFrom,
       XMPULong  numExpectedEntries);
void DeleteNameSpace(
       XMPNameSpace* theNameSpace);
XMPNameSpace*  HasNameSpace(
       XMPISOStr spaceName);
```

XMPSession

```
void Close();
XMPWindowState* GetWindowState();
```

```
XMPDispatcher* GetDispatcher();
XMPArbitrator* GetArbitrator();
XMPStorageSystem*  GetStorageSystem();
XMPClipboard* GetClipboard();
XMPDragAndDrop* GetDragAndDrop();
XMPSymbols* GetSymbols();
XMPMessageInterface* GetMessageInterface();
XMPNameResolver* GetNameResolver();
XMPTranslation* GetTranslation();
XMPUndo* GetUndo();
XMPTypeToken Tokenize(XMPType type);
void RemoveEntry(XMPType type);
XMPType GetType(XMPTypeToken token);
XMPSemanticInterface* GetShellSemtInterface();
XMPSemanticInterface* GetSemanticInterface();
```

XMPTransform

```
XMPTransform(
       XMPGraphicsSystem graphicsSystem,  XMPPlatformTransform
       platformTransform);
~XMPTransform();
XMPGraphicsSystem GetGraphicsSystem();
XMPPlatformTransform GetPlatformTransform();
void SetPlatformTransform(
       XMPGraphicsSystem graphicsSystem,  XMPPlatformTransform
       platformTransform);
XMPTransform* Reset();
XMPTransform* CopyFrom(
       XMPTransform* sourceTransform);
XMPTransform* Invert();
XMPBoolean IsSameAs(
       XMPTransform* compareTransform);
XMPTransform* PreCompose(
       XMPTransform* transform);
XMPTransform* PostCompose(
       XMPTransform* transform);
XMPPoint TransformPoint(XMPPoint point);
```

XMPTranslation

```
XMPBoolean GetTranslateMethod(
       XMPTypeSet* givenTypes,
       XMPOSType nativeType,
       XMPTranslateResult* result,
       XMPTranslateMethod* howToTranslate);
XMPTypeSet*  GetTranslationOf(
       XMPOSType fromType);
XMPPtr Translate(
       XMPStorageUnitView* storageUnit,  XMPTranslateMethod
       howToTranslate);
XMPPtr Translate(
       XMPPtr fromData,
       XMPOSType givenType,
       XMPTranslateMethod howToTranslate);
```

XMPTypeSet

```
XMPTypeSet();
~XMPTypeSet();
void AddType(XMPOSType type);
void RemoveType(XMPOSType type);
XMPBoolean ContainsType(XMPOSType type);
XMPULong GetTypeCount();
```

XMPUndo

```
void AddActionToHistory(
        XMPPart* whichPart,
        XMPActionData actionData,
        XMPActionType actionType,
        XMPName actionLabel);
void Undo();
void Redo();
void MarkActionHistory();
void ClearActionHistory(
        XMPRespectMarksChoices respectMarks);
void ClearRedoHistory();
void PeekUndoHistory(
        XMPPart** part,
        XMPActionData* actionData,
        XMPActionType* actionType,
        XMPName* actionLabel);
void PeekRedoHistory(
        XMPPart** part,
        XMPActionData* actionData,
        XMPActionType* actionType,
        XMPName* actionLabel);
```

XMPWindow

```
XMPPlatformWindow GetPlatformWindow();
XMPFrame* GetSourceFrame();
XMPCanvas* GetCanvas();
void SetCanvas(XMPCanvas* canvas);
XMPFrame* GetRootFrame();
void AddFrame(
        XMPFrame* newFrame,
        XMPFrame* siblingFrame,
        XMPFramePosition position);
void RemoveFrame(XMPFrame* oldFrame);
XMPFrame* GetFrameUnderPoint(XMPPoint aPoint);
void Close();
void Open();
void Internalize();
void Externalize();
XMPBoolean IsActive();
```

XMPWindowState

```
XMPWindow* GetXMPWindow(
        XMPPlatformWindow aWindow);
XMPWindow* GetActiveWindow();
XMPMenuBar GetBaseMenuBar();
```

```
void UpdateBaseMenus();
void SetMenuUpdateProc(
       XMPMenuUpdateProcPtr proc);
XMPMenuBar* GetCurrentMenuBar();
XMPMenuBar* CopyBaseMenuBar();
```

Scripting & semantic event processing

XMPSemanticInterface

```
void InstallObjectAccessor(
       DescType desiredClass,
       DescType containerType,
       XMPAccessorProcPtr theAccessor,
       XMPSLong accessorRefcon);
void GetObjectAccessor(
       DescType desiredClass,
       DescType containerType,
       XMPAccessorProcPtr* theAccessor,
       XMPSLong* accessorRefcon);
void  CallObjectAccessor(
       XMPPart*  thePart,
       DescType  desiredClass,
       XMPOSLToken* containerToken,
       DescType containerClass,
       DescType keyForm,
       AEDesc  keyData,
       XMPOSLToken* token);
void RemoveObjectAccessor(
       DescType desiredClass,
       DescType containerType,
       XMPAccessorProcPtr theAccessor);
void SetObjectCallbacks(
       XMPCompareProcPtr compareProc,
       XMPCountProcPtr countProc,
       XMPDisposeTokenProcPtr disposeTokenProc,
       XMPGetMarkTokenProcPtr getMarkTokenProc,
       XMPMarkProcPtr markProc,
       XMPAdjustMarksProcPtr adjustMarksProc,
       XMPPtr reserved);
void InstallEventHandler(
       AEEventClass theAEEventClass,
       AEEventID theAEEventID,
       XMPEventHandlerProcPtr handler,
       XMPSLong handlerRefcon);
void RemoveEventHandler(
       AEEventClass theAEEventClass,
       AEEventID theAEEventID,
       XMPEventHandlerProcPtr handler);
void GetEventHandler(
       AEEventClass theAEEventClass,
       AEEventID theAEEventID,
       XMPEventHandlerProcPtr* handler,
       XMPSLong* handlerRefcon);
void InstallCoercionHandler(
       DescType fromType,
       DescType toType,
       XMPCoercionHandler handler,
       XMPSLong handlerRefcon,
       XMPBoolean fromTypeIsDesc);
```

```
void RemoveCoercionHandler(
       DescType fromType,
       DescType toType,
       XMPCoercionHandler handler);
void GetCoercionHandler(
       DescType fromType,
       DescType toType,
       XMPCoercionHandler* handler,
       XMPSLong* handlerRefcon,
       XMPBoolean* fromTypeIsDesc);
void RemoveSpecialHandler(
       AEKeyword functionClass,
       XMPEventHandlerProcPtr handler);
void GetSpecialHandler(
       AEKeyword functionClass,
       XMPEventHandlerProcPtr* handler);
```

XMPMessageInterface

```
XMPSShort  CreateEvent(
       AEEventClass theAEEventClass,
       AEEventID  theAEEventID,
       AEAddressDesc* target,
       XMPSLong  transactionID,
       AppleEvent*  result);
void  Send(
       XMPPart*  thePart,
       AppleEvent*  theAppleEvent,
       AppleEvent*  reply,
       AESendMode  sendMode,
       AESendPriority  sendPriority,
       XMPULong  timeOutInTicks,
       XMPIdleProcPtr  idleProc,
       XMPEventFilterProcPtr filterProc);
XMPBoolean ProcessSemanticEvent(
       const XMPEventData theEvent);
```

XMPNameResolver

```
XMPPart* Resolve(
       AEDesc*  theObject,
       XMPSShort  callbackFlags,
       XMPOSLToken* token);
XMPOSLToken* CreatePartToken(XMPPart* aPart);
XMPOSLToken* CreateToken(
       DescType  type,
       XMPPart*  inPart,
       XMPPtr  data);
void  DisposeToken(XMPOSLToken* theToken);
```

Extended event dispatch control

These calls are available to the part developer, but generally uninteresting unless some form of advanced event dispatching is required. In this case, the part developer has to implement XMPFocusModule and/or XMPDispatchModule objects and install them in the arbitrator and dispatcher respectively.

XMPArbitrator

```
void RegisterFocus(
       XMPTypeToken focus,
       XMPFocusModule* focusModule);
```

```
void UnregisterFocus(XMPTypeToken focus);
```

XMPFocusModule

```
XMPBoolean IsFocusExclusive(
       XMPTypeToken focus)
void SetFocusOwnership(
       XMPTypeToken focus,
       XMPFrame* frame)
void UnsetFocusOwnership(
       XMPTypeToken focus,
       XMPFrame* frame)
XMPFrame* GetFocusOwner(XMPTypeToken focus)
XMPFocusOwnerIterator* CreateOwnerIterator(XMPTypeToken focus)
```

XMPDispatcher

```
void AddDispatchModule(
       XMPDispatchCode code,
       XMPDispatchModule* dispatchModule,
       XMPBoolean monitor);
void RemoveDispatchModule(XMPDispatchCode code);
```

XMPDispatchModule

```
XMPDispatchModule(XMPSession* session);
~XMPDispatchModule();
XMPBoolean Dispatch(XMPEventData event);
```

Extended draft control

These calls are available to the part developer, but generally uninteresting unless some form of special control over drafts is desired.

XMPDocument

```
XMPDocument* CollapseDrafts(
       XMPDraft* from,
       XMPDraft* to);
XMPDraft* GetDraft(
       XMPDraftPermissions perms,
       XMPDraftID id,
       XMPDraft* draft,
       XMPPositionCode posCode,
       XMPBoolean release);
XMPDraft* GetBaseDraft(
       XMPDraftPermissions perms);
XMPDraft* CreateDraft(
       XMPDraft* below,
       XMPBoolean releaseBelow);
XMPDocument* ReleaseDraft(XMPDraft* draft);
void SaveToAPrevDraft(
       XMPDraft* from,
       XMPDraft* to);
XMPDraft* SetBaseDraftFromForeignDraft(
       XMPDraft* draft);
```

Container application implementors

These APIs are primarily interesting to the developer of a shell application or those retrofitting existing applications to contain OpenDoc parts.

XMPContainer

```
XMPStorageSystem* GetStorageSystem()
XMPContainerID GetID();
XMPContainerName GetName()
void SetName(XMPContainerName name)
void Release()
XMPDocument* GetDocument(XMPDocumentID id);
```

XMPDocument

```
XMPContainer* GetContainer();
XMPDocumentID GetID();
XMPDocumentName GetName();
void SetName(XMPDocumentName name);
void Release();
```