OMG RFP Submission

Compound Presentation and Compound Interchange Facilities

Part I

Apple Computer, Incorporated

Component Integration Laboratories, Incorporated

International Business Machines Corporation

Novell, Incorporated

December 13, 1995

Abstract

The OMG Common Facilities RFP1 provides for two facilities; a Compound Presentation Facility, and a Compound Interchange Facility. The RFP recognizes that these two facilities may be tightly bound, and therefore allows both facilities to be included in a single response.

OpenDoc is a technology that encompasses both facilities. This document details the architecture and the associated class library expressed in IDL. Because OpenDoc is a response to the first OMG Common Facilities RFP, no other common facilities exist to support the Common Presentation and Common Interchange facilities. As a result, two additional facilities are included in this submission; the Common Core facility and the Common Utility facility. These additional facilities make it possible for the classes to be partioned in the most appropriate manner.

Table Of Contents

List Of Figures

List of Tables

1. Introduction

This document is Volume I of the response to the OMG Common Facilities RFP1.

1.1 Submission contacts

All questions regarding this submission should be directed to:

Mark Minshull
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
USA
phone: +1 408 974-7677
email: mark_minshull@powertalk.apple.com

Richard Hoffman
IBM Corporation
M/S 9356
11400 Burnet Road
Austin, TX 78758
USA
phone: +1 512 838-0732
email: ubiquity@ausmvm.vnet.ibm.com

Mark Ericson
Novell, Inc.
MS: C-11-2
122 E 1700S
Provo, UT 84606-6194
USA
phone: +1 801 429 3728
email: marke@novell.com

David Berkowitz
Component Integration Laboratories
P.O. Box 61747
Sunnyvale, CA 94088-1747
USA
phone: +1 408 864 0300
email: david@cil.org

1.2 Organization of this document

This submission is organized into two volumes because of its size. The first volume provides overview and roadmap information. It also details how the submission meets the various requirements of the RFP.

The second volume contains the detailed description of the facilities.

1.3 References

Object Management Architecture Guide, Revision 2.0, Second Edition, September 1, 1992. OMG TC Document 92-11-1

The Common Object Request Broker: Architecture and Specification, Revision 1.2. OMG TC Document 93-12-43

Common Facilities Request For Information, February 1994. OMG TC 94-2-11
Common Facilities Architecture, August 1994. OMG TC Document 94-8-0.
Object Services Architecture, Revision 1.0, October, 1992. OMG TC Document 92-8-4
Object Services Roadmap, October, 1992. OMG TC Document 92-8-10
Common Object Services Specification, Volume 1. OMG TC Document 94-1-1

2. Overview

The facilities defined in this standard enable the creation of cooperative component software that supports compound documents, that can be customized, that can be used collaboratively, and that is available across multiple platforms.

Although it is possible to discuss a compound presentation facility independently of a compound interchange facility, such a discussion would be difficult to follow. Therefore, the discussion of these facilities, plus the compound core and compound utility facilities, is combined except in the actual definitions of the facilities. In general, elements involved in storage and data transfer belong to the compound interchange facility; everything else is part of the compound presentation facility.

The four facilities are collectively referred to as OpenDoc, which is the name of the original project from which these facilities are derived.

2.1 Rationale

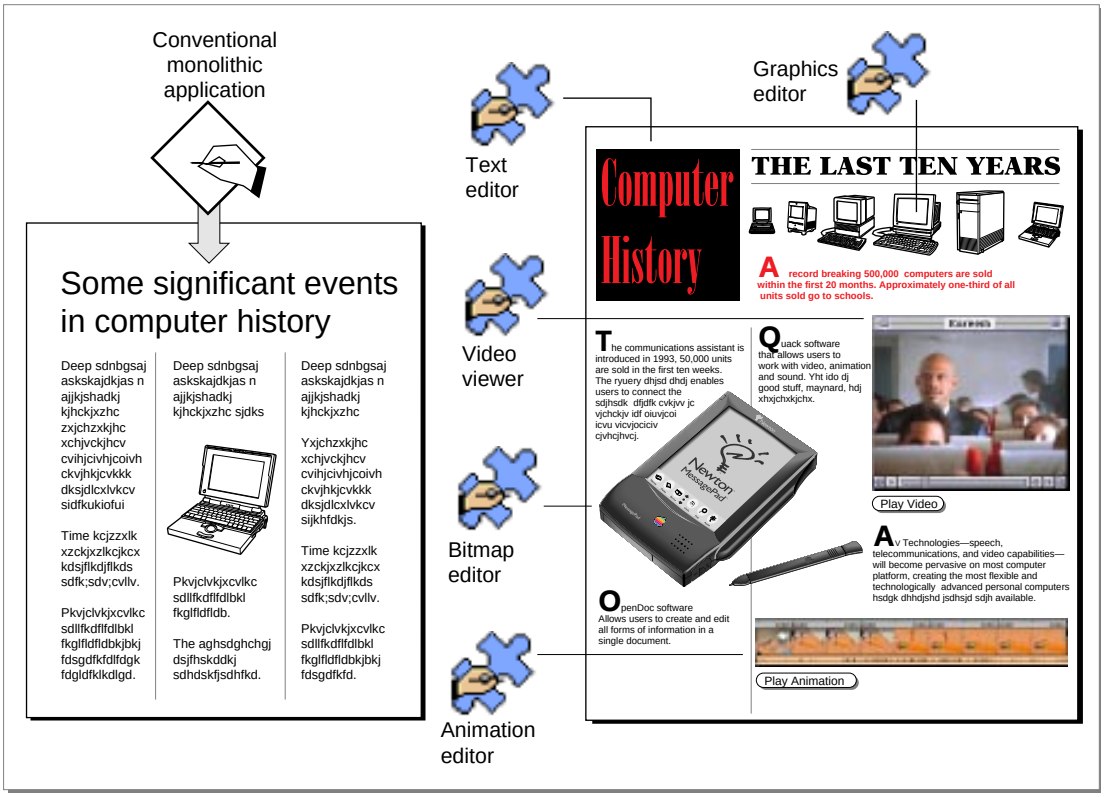
Customer demand for increasingly sophisticated and integrated software solutions has led to large and sometimes unwieldy application packages, feature-laden but difficult to maintain and modify. Developing, maintaining, and upgrading these large, cumbersome applications can require a vast organization; the programs are difficult to create, they are expensive to maintain, and they can take years to revise.

Upgrades or bug-fixes in one component of such an application require the developer to release—and the customer to buy—a complete new version of the entire package. Some developers have added extension mechanisms to their packages to allow addition or replacement of certain components, but the extensions are proprietary, incompatible with other applications, and applicable only to certain parts of the package

Because of the barriers put up by the current application architecture, users are often frustrated in attempting to perform common tasks, especially those than require the features of multiple, independent application packages.

OpenDoc addresses these issues by enabling the development of a new kind of application that provides advantages to both users and developers. OpenDoc replaces the architecture of conventional **monolithic applications**, in which a single software package is responsible for all of its documents' contents, with one of **components**, in which each software module edits its own content, no matter what document that content may be in. See Figure 1.

Figure 1 Monolithic application vs. components



OpenDoc allows application developers to take a modular approach to development and maintenance. Its component-software architecture can make the design, development, testing, and marketing of integrated software packages easier and more reliable as compared to their monolithic counterparts. Developers can make incremental changes to component products without affecting unrelated components, and can therefore get those changes to users far more rapidly than is possible using the monolithic approach.

OpenDoc components can allow users to assemble customized **compound documents** out of diverse types of data. They can also support cross-platform sharing of information. They resolve the user frustrations that occur at the boundaries between conventional, monolithic applications.

Table 1 summarizes some of the principal advantages of the OpenDoc approach to software, for both users and developers.

Table 1 OpenDoc Advantages

	for Users	for Software Engineering	for Software Marketing	for Development Teams
Components are…				
…Modular	Easy to add or replace document parts	Easy to upgrade components. Easier to test components	Great flexibility in assembling packages	Can create a component that works seamlessly with all others
…Small	Code for individual components takes up much less disk space and memory	Easier to design, code, debug, and test	Easier, cheaper distribution	Faster development, easier distribution of a component
…Cross-platform	Documents travel across platforms; users select familiar editors on each	Can leverage development effort on one platform to all others	Opportunities for increased market share	Application of limited resources can be used across all platforms

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

- Compound document support
  OpenDoc includes a set of interoperability protocols that allows code produced by independent development teams to cooperate to produce a single document for the end user. 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 are provided.
- Multiple Revision Support
  OpenDoc documents must have the ability to be passed through review cycles with minimal pain on the part of users. OpenDoc includes a draft capability which allows work to be performed on multiple versions of a document.
- Cross-platform
  OpenDoc is designed as a cross-platform architecture.
- Replaceability
  OpenDoc allows the replacement of the implementation of any of its subsystems on any platform. All meta-data for persistent storage is fully documented so that interoperability is maintained.
- Parsimony
  OpenDoc does not specify drawing systems, coordinate systems, window systems, human interface guidelines, or many other platform specific elements. This makes the architecture more generally available for use across platforms while standards for these other areas are developed.

2.2 Architectural Model

OpenDoc is an architecture that facilitates the construction of compound, customizable, collaborative, and cross-platform documents. To do this, OpenDoc uses a *document–centered* user model instead of an *application–centered* one. The user focuses on constructing a document or performing an individual task, rather than using any particular application. The software that manipulates a document is hidden, allowing users to manipulate the parts of the document without having to invoke or switch applications.

This does not mean that OpenDoc supports only those kinds of data found in paper documents; an OpenDoc document can contain data as diverse as navigable movies, sounds, animation, database information such as networked calendars or virtual folders, as well as traditional spreadsheets, graphics, and text. OpenDoc is an ideal architecture for multimedia documents. In OpenDoc, each new kind of medium that is developed—video, sound, animation, simulation, and so on—can be represented as a part of any document. Thus, an OpenDoc document is automatically able to contain future kinds of media, even kinds not yet envisioned, without any modification.

Although OpenDoc lends itself directly to complex and sophisticated layout, its usefulness is by no means restricted to page-layout kinds of applications or even compound documents. Tools such as spell-checkers can be created as components, and can then access the contents of any parts in a document that support them; database-access components can feed information to any parts of a document; larger programs such as high-end printing applications can use specialized components to manipulate the data of all parts of a document for purposes such as proof printing and color matching.

The rest of this section summarizes the main features of OpenDoc, for both users and developers.

2.2.1 Parts and Documents

OpenDoc uses a few simple ideas to create a structure that integrates a wide range of capabilities. *Documents,* their *parts,* their *frames,* and the *part-editor* code that manipulates them form the basics of OpenDoc. These elements are manifested in a set of object-oriented classes. These classes define a number of interoperability protocols, implemented as object classes, that allow independently developed software components to cooperate in producing a single document for the end user. Through the classes, these cooperating components share user-interface resources, negotiate over document layout on screen and on printing devices, share storage containers, and create data links to one another.

This section describes what part editors, parts, and frames are, and how parts are categorized, drawn, and stored in documents. It also describes how part editors relate to other kinds of OpenDoc components, such those that simply display parts and those that provide services to parts.

2.2.1.1 Parts and Part Editors

In OpenDoc documents, application components called **part editors** replace conventional monolithic applications. Each part editor is responsible for manipulating data of a single kind in a document. The user does not directly invoke part editors, however. The user works with **document parts** (or just **parts**), the pieces of a document that include both the document data and a mechanism that invokes the part-editor code that manipulates it.

Figure 2 Parts in an OpenDoc document

A page as it normally appears

A page with part boundaries shown

In general, a user can perform all the tasks of an application by manipulating a part instead of separately invoking its part editor. For example, a user can create and use a spreadsheet part that looks and acts just like a spreadsheet created by a spreadsheet application. However, there are four main differences between a document consisting of parts and a document created by a conventional application:

- The document is built and manipulated differently. OpenDoc users can assemble a document out of parts of any kind—using their graphics part editor of choice, for example, to embed illustrations within any word processor document.
- New parts of any kind can be added to the document. The user can add additional part editors—for example, a charting utility to accompany a spreadsheet—and use them to embed charts into existing documents.
- In editing, copying, or pasting a part, the user need not be aware of which part editor is executing. The user directly manipulates the part itself, within the context of the document. (It is not necessary to open the part into a separate window for editing.) The document containing that part is opened and closed independently of the part's part editor.
- The user can replace part editors. If the editor for a specific kind of part in a document is not available, or if the user prefers to use a different part editor—for example, to replace one charting utility with another—the user can specify that the new editor be used with all parts created under the previous editor. This gives a user the freedom to work with all of the editors of a package, or replace any of them with others that the user prefers.

A part editor has fewer responsibilities than a conventional application. Each part editor must

- display its part, both on screen and when printing
- edit its part, by changing the state of the part in response to events caused by user actions
- store its part, both persistently and at run time

A part is the equivalent of an object-oriented programmatic object, in that it encapsulates both state and behavior. The part data provides the state information, and the part editor provides the behavior; when bound together, they form an editable object.

Part editors are dynamically associated with their parts at run time, based on the kinds of data that the part contains. This delivers a smooth user experience because any sort of part might appear in any document at any time.

2.2.1.1.1 Editors, Viewers, and Other Components

There are two flavors of application components that a user can apply to compound documents: part editors and part viewers. A part editor, as noted earlier, is a full-featured application component; it allows the creation, editing, and viewing of parts of a particular kind. Part editors are the functional replacements for conventional applications; they represent the developer's primary investment. Like applications, part editors are sold or licensed, and are legally protected from unauthorized copying and distribution.

A **part viewer** is a special variety of part editor that can display and print a part of a particular kind, but cannot be used to create or even edit such a part. To enhance the portability of OpenDoc compound documents across machines and across platforms, it is important that part viewers of all kinds be widely available. Developers are expected to create and freely distribute part viewers without restriction, for all kinds of parts that they support. A part viewer is really just a part editor with its editing and part-creation capability removed; the developer can create both from the same code base.

OpenDoc also supports the development of components that, unlike part editors and part viewers, are not directly involved in creating or displaying document parts. Spell-checking or database-access tools, when developed as service components, can increase the capabilities of part editors, and can be applied to a single part, all the parts of a document, or even across separate documents.

2.2.1.1.2 Frames and Embedded Parts

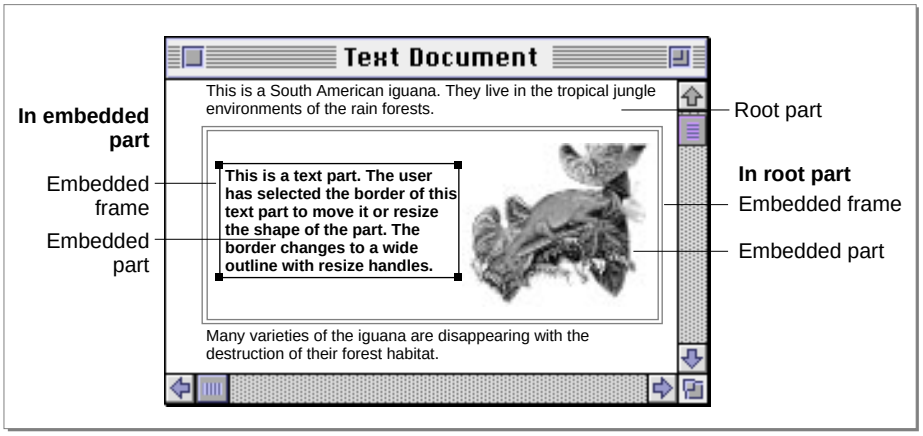
Parts in an OpenDoc document have a hierarchical arrangement with each other. The logical structure of a document, which underlies its graphical presentation, consists of the **embedding** relationships among its parts and their frames.

Parts are displayed in **frames.** The **display frame** of one part—the frame within which the part is viewed—can itself be embedded within the content of a second part. In that case the frame is an **embedded frame** of the second part, and the second part is the **containing part** of the first part. (Conversely, the first part is an **embedded part** of the second part.)

An embedded part is logically contained in its containing part, just as its frame is visually embedded in the containing part's frame. A containing part treats its embedded frames as regular elements of its own content. The containing part can move, select, delete, or otherwise manipulate the embedded frame. The embedded part itself, however, takes care of drawing and event-handling within that frame.

Every document has a single part at its top level, the **root part,** in which all other parts in the document are directly or indirectly embedded. The root part controls the basic layout structure of the document (such as text or graphics) and the document's overall printing behavior. Figure 3 shows an example of the relationships of the root part and two embedded parts.

Figure 3 Embedded parts and frames



Because it can now contain embedded frames displaying any kind of content, the document is no longer a monolithic block of data under the control of a single application, but is instead composed of many smaller blocks of content controlled by many smaller software components. In large part, OpenDoc exists to provide the protocols that keep the components from getting in each another's way at run time and to keep documents editable and uncorrupted.

All parts can be embedded in other parts, and all parts must be able to function as the root part of a document. However, not all parts are required to be able to contain other parts; simple or specialized utilities such as clocks or sound players may have no reason to embed other parts. Such parts are called **noncontainer parts;** they can be embedded in any part, but they cannot embed other parts within themselves. Parts that can embed as well as be embedded, on the other hand, are called **container parts.** It is somewhat simpler to write an editor for a noncontainer part than for a container part, although container parts provide a far more general and flexible user experience. It is recommended that editors be container parts unless embedding makes absolutely no sense for an editor.

Parts can have more than one frame. A part embedded in a document is not restricted to appearing in a single frame. Parts can have multiple frames, displaying either duplicate views or different aspects of the same part. See “Presentation” on page 13 for more information.

2.2.1.2 Part Data Types

Fundamental to the idea of a compound document is that it can hold different types of data. Each part editor can manipulate only its own kinds of data, called its **intrinsic content.** For a simple drawing part, for example, the elements of its intrinsic content may be bitmaps; for a simple text part, they may be characters, words, and paragraphs. If a part contains an embedded part, that **embedded content** is manipulated by its own part editor. No part editor is asked to manipulate the content elements of another kind of part.

OpenDoc classifies those types of intrinsic content, in order to assign the correct part editor to the data of each part, and in order to facilitate the translation and transfer of data from one part to another.

2.2.1.2.1 Part Kind

Different parts in a document hold data of different purpose and different format, understandable to different part editors. OpenDoc must be able to associate a part editor only with data that the editor can properly manipulate, in order to avoid file corruption and to have meaningful drawing and editing.

OpenDoc uses the concept of **part kind,** a typing scheme analogous to file type, to determine what part editor is to be associated with a given part in a document. Because OpenDoc documents are not associated with any single application, a file type is insufficient in this case; each part within a document needs its own “type,” or in this case, part kind.

Part kinds are specified as **ISO strings**, (null-terminated 7-bit ASCII strings), although they are usually manipulated as **tokens** (short, regularized representations of those strings) by OpenDoc and by part editors. A part kind specifies the exact data format manipulated by an editor; it may have a name similar to the editor name, such as “SurfDraw” or “WindBase III”, although names more descriptive of the data format itself (such as “plain text” or “QuickTime movie”) are preferable.

Part-editor developers define the part kinds of their own editors. A single editor can manipulate more than one part kind, if it is designed to do so, and a single part can be stored with multiple representations of its data, each of a different part kind. All parts created by an editor initially have its part kind (or kinds), although users can assign new kinds to a part; see “Translation” on page 10.

OMG will ensure that a registry of part kinds is maintained.

2.2.1.2.2 Part Category

The concept of part kind does not consider that many data formats have similar purpose and may have very similar formats. For example, data stored as plain ASCII text could conceivably be manipulated by any of a number of part editors. Likewise, video data stored according to a standard might be manipulated by many different video editors. Furthermore, text that is close to pure ASCII, and stored video that almost but not exactly follows a particular standard, might nevertheless be editable or displayable to some extent by an ASCII editor or standard video editor.

To restrict the editing or display of a part of a given kind to the exact part editor that actually created it can be unduly confining; unless the user has every part editor that created a document, the user cannot edit or even view all parts of it.

To promote the wide distribution and easy sharing of documents, OpenDoc gives users the flexibility of substituting new part editors for those used to create any of the parts of a document. OpenDoc does this by defining **part category,** a general description of the kind of data manipulated by a part editor. When users speak of a “text part” or a “graphics part,” they are using an informal designation of part category.

Like part kinds, part categories are specified as ISO strings and manipulated as tokens. Part categories have broad designations, such as “text”, “styled text”, “bitmap”, or “relational database”. Each part editor specifies the part categories that it can manipulate. OpenDoc uses part category to help assign part editors to parts for which the preferred editor is not present. The user cannot directly change the part category (or categories) of a part.

When the user opens a document (or when the user drops or pastes data) containing a part of a kind for which the user has no preferred part editor, OpenDoc gives the user the option of substituting another editor, of the same part category, that can edit that part kind. If the user chooses another part editor, the part is then assigned the part kind of that new editor. If the user chooses not to substitute part editors, or if there is no appropriate editor available, the part remains unviewable and uneditable, and its part kind does not change. The user may, however, be given the option of translating the data into an editable format, as described next.

If there is no available editor that can display a part, OpenDoc displays an outline of the part’s frame. Lack of a part editor never prevents a user from opening a document.

OMG will ensure that a registry of part categories is maintained.

2.2.1.2.3 Translation

Changing the part editor for a part often means changing data formats and may involve loss of information. For example, any text editor may be able to display and edit ASCII text without loss, but a sophisticated word processor usually needs a complex software filter to be able to read another word processor’s data, and the results may not come across perfectly.

When the user chooses to use a part editor with a part whose part kinds are not directly usable by the editor, OpenDoc may perform the necessary **translation** to convert the part from one part kind to another that the editor can make use of. The possible translations between part kinds depends on what part editors are available on the user’s machine; the **fidelity,** or quality, of the translation depends on the presence and sophistication of the **translators,** or filters, that perform the translation. Translators are platform-specific utilities that are independent of OpenDoc; OpenDoc simply provides an object wrapper for a given platform’s translation facilities.

When OpenDoc translates an embedded part so that its part kind becomes the same as that of its containing part, there is usually no reason for the data to remain as a separate, embedded part. In that case the containing part usually incorporates the information into its intrinsic content.

OpenDoc also provides support for translation when a document is first opened (see “Binding with Translation” on page 56), during Clipboard transfers, drag-and-drop, and linking (see “Data Transfer” on page 18).

2.2.1.3 Displaying Parts

In preparing a compound document for viewing, two issues are of prime importance: managing the competition for space among the parts of the document, and making provisions for each part to draw itself.

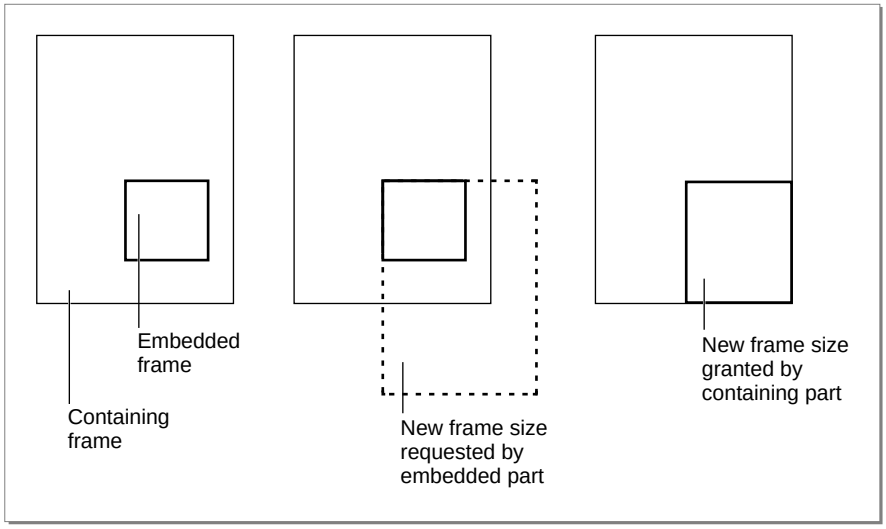
Although OpenDoc provides a platform-independent interface for part editors it does not include any drawing commands or detailed graphics structures. OpenDoc provides an environment for managing the geometric relationships among frames, and wrappers for accessing platform-specific graphic and imaging structures.

2.2.1.3.1 Frame Negotiation

In a compound document with embedded parts, the embedding hierarchy and the frame locations determine the geometric relationships among parts. Each part controls the positions, sizes, and shapes of the frames embedded within it. If an embedded part needs to change the size of its frame or add another frame, it must negotiate for that change with its containing part. This **frame negotiation** allows an embedded part to communicate its needs to its containing part; however, the containing part has ultimate control over the embedded frames’ sizes and shapes.

Figure 4 shows a simple example of frame negotiation. A user edits an embedded part, adding enough data so that the content no longer fits in the embedded part’s current frame size. The embedded part requests a larger frame size from the containing part. The containing part can either grant the request or return a different frame size from that requested. In this example, the containing part cannot accommodate the full size of the requested frame, and so returns a frame size to the embedded part that is larger than the previous frame size but not as large as the requested size.

Figure 4 Frame negotiation



2.2.1.3.2 Drawing Structures

Each part in an OpenDoc document is responsible for drawing its own content (including, at certain times, the borders of the frames embedded within it). What that means, of course, is that a part does *not* draw the interiors of its embedded frames, because they contain the content of other parts (which must draw themselves). It also means that drawing a document is a cooperative effort, for which no part editor is completely responsible; each part editor is notified by OpenDoc when it must draw its own part.

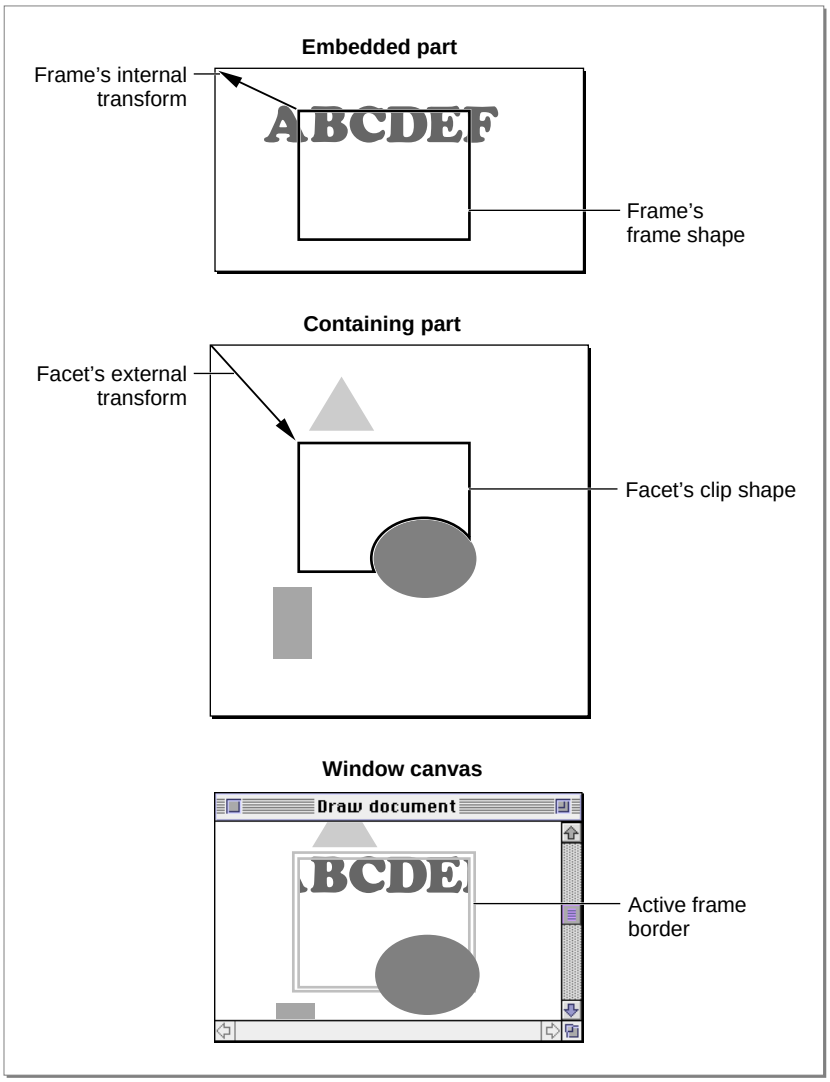
Drawing in OpenDoc relies on three fundamental graphics-system-specific structures, for which OpenDoc provides wrapper objects:

- **canvas,** a description of a drawing environment.
- **shape,** a structure that describes a geometric shape.
- **transform,** a structure that describes an offset plus a scaling factor or other geometric transformation.

In addition to these structures, when it comes to actually drawing a part within a frame, OpenDoc relies on the facet, an object that is closely related to the frame. A **facet** is an object that represents where a frame is drawn on a canvas. A frame might typically have a single facet, but, as in the case of off screen buffering, split views of a part, or creation of special graphic effects, there may be more than one facet per frame.

In general, frames control the geometric relationships of the content that they display, whereas facets control the geometric relationships of frames with their containing frame or window. The facet is associated with a specific drawing canvas, and both frames and facets have associated shapes and transforms. Figure 5 summarizes some of the basic relationships among frames, facets, canvas, shapes, and transforms in drawing.

Figure 5 Frames and Facets in Drawing



The top of Figure 5 shows the content area of an embedded part as a page with text on it. The portion of the part that is available for drawing is the portion within the **frame shape,** assigned to the part by its containing part. (A part may have more than one frame, but only one is shown here). The frame's **internal transform** positions the frame over the part content. (Another shape, the **used shape,** defines what portion of the area of the frame shape is actually drawn to; in this case, the used shape equals the frame shape.)

The center of Figure 5 shows the facet associated with this embedded part's frame. (A frame may have more than one facet, but only one is shown here). The **clip shape** of the facet defines where drawing can occur, in relation to the content of the containing part. In this case, the containing part is the root part in a document window, but it could be an embedded part displayed in its own frame. The clip shape is typically similar to the frame shape, except that, as shown in Figure 5, it may have additional clipping to account for other embedded parts or for elements of the containing part that overlap it. The facet's **external transform** positions the facet within the containing part's frame and facet. (Another shape, the **active shape,** defines what portion of the area of the facet the embedded part will respond to events within; in this case, the active shape equals the frame shape.)

The bottom of Figure 5 shows the results; the portion of the embedded part defined by the frame is drawn in the area defined by the facet. If this embedded part is the **active part,** meaning that the user can edit its contents, OpenDoc draws a **frame border** around it; the shape of that border equals the facet's active shape.

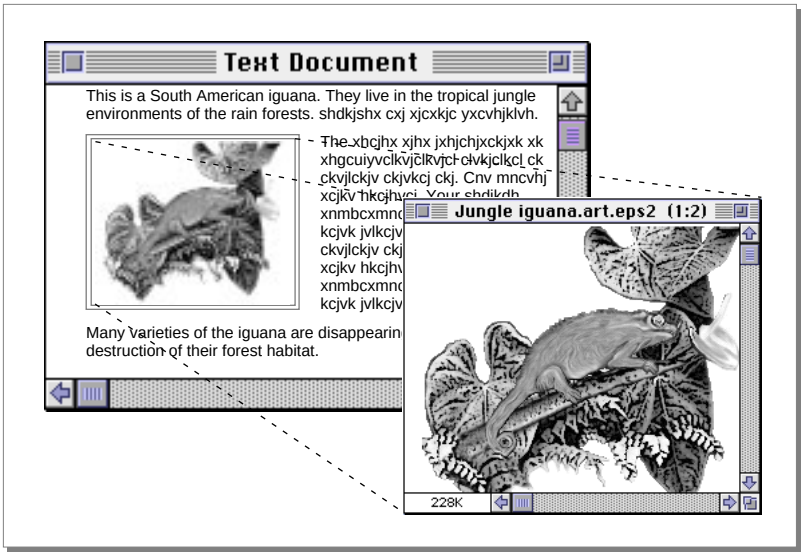
2.2.1.3.3 Document Windows and Part Windows

Compound documents are displayed in windows. A **document window** is a window that holds a single OpenDoc document; that document can contain a single part or many parts. Document windows in OpenDoc are essentially the same as the windows that display a conventional application's documents.

An architectural cornerstone of OpenDoc is that it provides **in-place editing** of all parts in a compound document. Users can manipulate the content of any part, no matter how deeply embedded, directly within the frame that displays the part.

Nevertheless, a user might wish to view more of a part than is displayed within a frame. Even if a frame is resizable and supports scrolling of its contents, it may be more convenient to view that frame's part separately, in its own window. OpenDoc supports this by allowing users to open a separate window, called a **part window,** that looks similar to a document window. See Figure 6, for example.

Figure 6 A framed part opened up into a part window



Part windows have a slightly different appearance and behavior from document windows. Furthermore, because a part editor controls all of the content of a part window, it takes a more active role in window handling than it does for document windows. See "Windows" on page 37 for information on handling part windows.

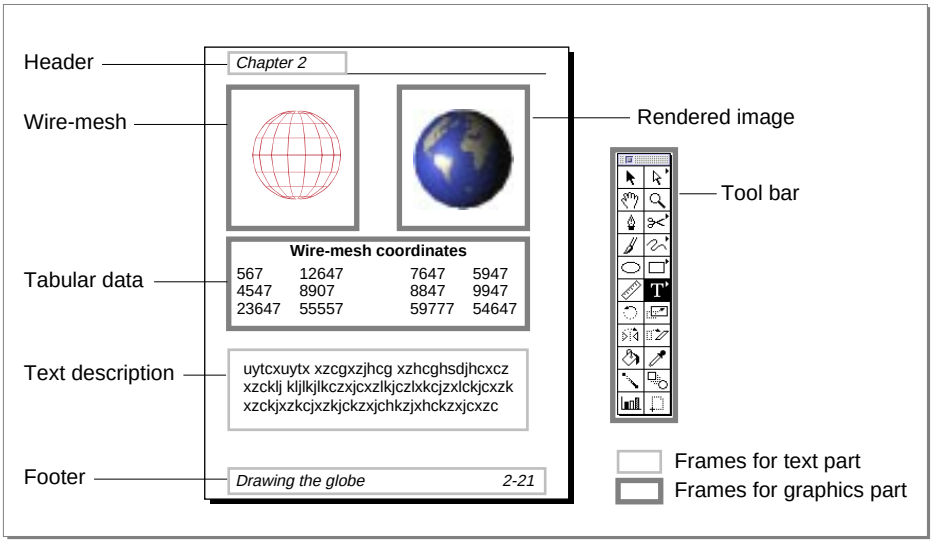
2.2.1.3.4 Presentation

There are many ways that a part editor can draw the contents of a part. A word processor can draw its data as plain text, styled text, or complete page layouts; a spreadsheet can draw its data as text, tables, graphs, or charts; a 3D drawing program can draw it shapes as wire-mesh polyhedrons, filled polyhedrons, or surface-rendered shapes with specified lighting parameters.

For any kind of program, individual frames of a single part can display different portions or different views of its data. For example, in a document processing part, one frame of a page could display the header or footer, while another could display the text of the page, and yet another could show the outline of the document. For a 3D graphics part, different frames could show different views (top, front, side) of the same object. For any kind of program, an auxiliary palette, tool bar, or other set of controls might be placed in a separate frame and be considered an alternative "view" of the part to which it applies.

OpenDoc calls such different part-display aspects part **presentations,** and imposes no restrictions on them. Figure 7 shows some examples of different presentations for individual parts.

Figure 7 Several presentations for two parts in a document



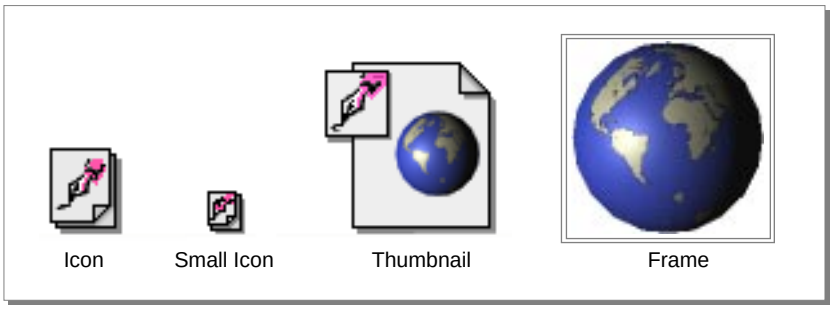
A part editor determines the presentations that it is capable of, and stores a presentation designation (for use by its own drawing functions) in each of the part's frames. Presentation, as well as other display-related information, can be stored in the **part info** property of a frame. Each frame and each facet has a part info property that can be used to hold private part-editor data.

2.2.1.3.5 View Type

OpenDoc does not specify presentation types, but it nevertheless defines some fundamental aspects of part display. Each frame has a **view type,** a designation of the overall display form of the part in that frame. Basically, view type states whether a part is to be displayed in icon form or with its contents visible.

In most situations in most documents, each part displays its contents, or some portion of its contents, within its frame. However, a part can also display itself as one of several kinds of icons; see Figure 8 for examples. Each basic display form (standard icon, small icon, thumbnail icon, or framed) is a separate view type; part editors store a designation of the part's view type in each of its display frames. A single part can of course have different view types in different frames.

Figure 8 View types for a part



On the desktop, in file folders, and in a parts bin (see "Document-Construction Aids" on page 16), parts are individual closed documents and by default have an iconic view type. When such a document is opened, the part gives its new frame a new (framed) view type, and its contents (including embedded parts) then become visible. Nevertheless, opened documents can have parts in them with iconic view types, and a part with an iconic view type is not necessarily closed or inactive. A sound-playing part, for example, might have nothing but an iconic view type.

Each containing part can specify the view type it prefers an embedded part to have by setting a value in the embedded part's frame. For example, file folders and the desktop usually prefer embedded parts to have an iconic view type; parts in documents, on the other hand, usually prefer their embedded parts to appear in frames. Containing parts can

specify the view type for each embedded frame they create; embedded parts should initially display themselves in the view type expected by their containing parts.

2.2.1.4 Document Storage

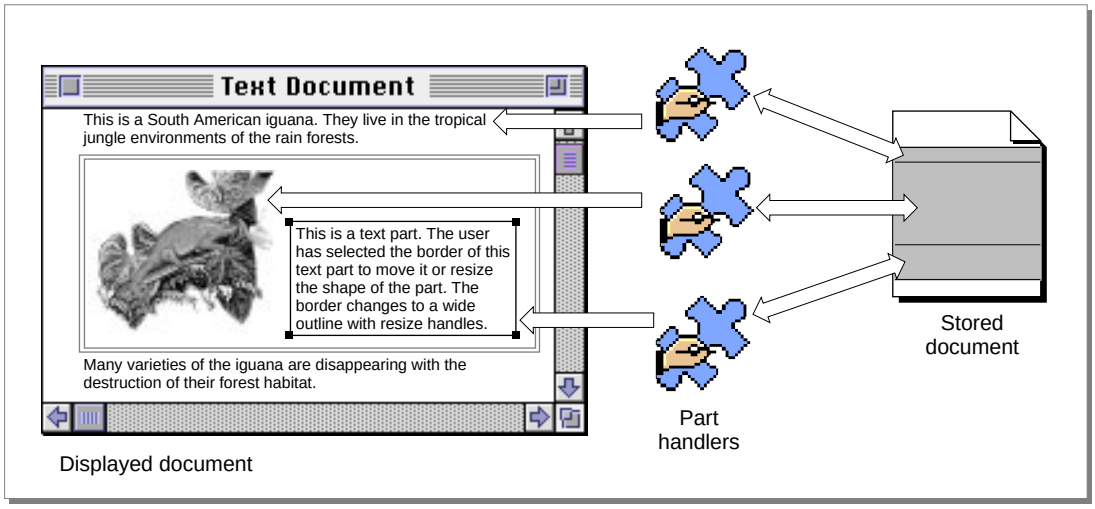
All the data of all parts in a document, plus all information about frames and embedding, is stored in a single document file. Rather than requiring the user to manually manage the various file formats that make up a compound document, OpenDoc holds all the pieces. This makes storage easier for developers, and exchanging documents easier for users.

2.2.1.4.1 Storage Basics

The OpenDoc storage system manages persistent storage for parts. It is a high-level persistent storage mechanism that enables multiple part editors to share a single document file effectively. The storage system is implemented on top of the native storage facilities for each platform that supports OpenDoc.

Storage in OpenDoc does not use an object-oriented database, but instead uses a system of structured elements, each of which can contain many data streams. The system effectively gives each part its own storage stream, as shown in Figure 9. This design maintains maximum compatibility with the many existing application storage systems that assume stream-based I/O.

Figure 9 Multiple data streams in a single OpenDoc document



The core of the OpenDoc storage system is the **storage unit,** an element that can contain one or more data streams. The data of each part in a document is kept in at least one storage unit, distinct from other parts' data. Storage units can also include references to other storage units, and OpenDoc uses chains of such references to store the embedding relationships of the parts within a document.

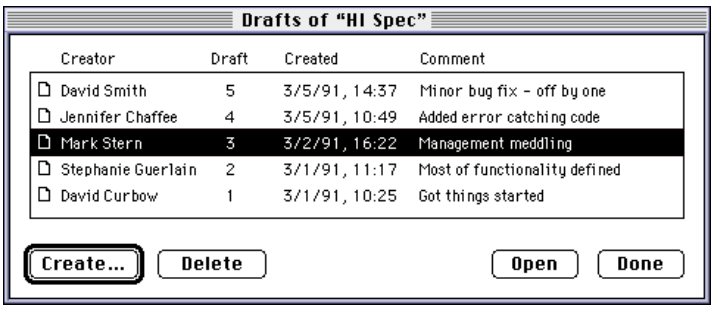
The OpenDoc storage system is not just for documents. Other types of data interchange make use of storage units; see "Data Transfer" on page 18.

2.2.1.4.2 Document Drafts

OpenDoc documents have a history that can be preserved and inspected through the mechanism of drafts. A **draft** is a captured record of the state of a document at a given time; the user decides when to create a new draft of the document, and when to delete older drafts. All drafts are stored together in the same file, with no redundantly stored data.

The OpenDoc draft mechanism helps in the shared creation of documents. When multiple users share a document, each can in turn save the current state of the document as a draft, and then make any desired changes. Users can always look back through the drafts of the document they and the others have created. Also, if translation occurs during the process of sharing documents, the user can consult an older draft to regain access to formatting information that might have been lost in translation. Figure 10 shows an example of a dialog box through which the user can manipulate drafts.

Figure 10 The Draft History dialog box

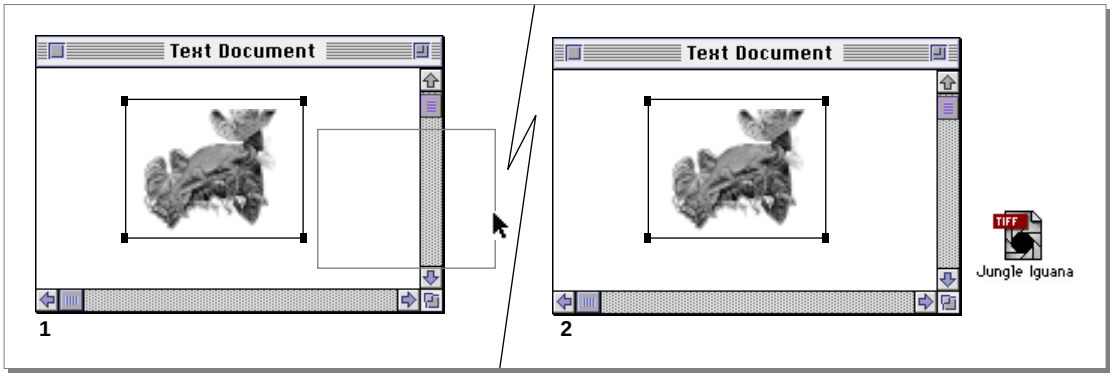


2.2.1.4.3 Document-Construction Aids

OpenDoc provides an additional aid to the user for the construction of compound documents. Specialize parts called **Stationery** exist whose only purpose is to serve as templates for the creation of other parts. A stationery part is never opened; when the user attempts to open a stationery part, a copy of that part is created and opened instead. Users can create stationery with specific formatting and content, to create letterhead, forms, or other templates.

Furthermore, as far as user experience is concerned, OpenDoc blurs the distinction between a single part and an entire document. If the user, employing either drag-and-drop or Clipboard transfer, moves a closed document (represented as an icon on the desktop) into an open document window, a copy of the transferred document immediately becomes an embedded part (or set of parts, if the closed document contained more than one part) in the window's document. Likewise, if the user selects the frame of an embedded part in an open document window and drags or otherwise moves that part to the desktop, a copy of that part immediately becomes a separate, closed document represented by icon, as shown in Figure 11.

Figure 11 Dragging a framed part to the desktop



2.2.2 Event-Handling

Most part editors interact with the user primarily by responding to **user events,** the messages sent or posted by the operating system in response to user actions, or to activation or deactivation of a part, or to messages from other event sources. Based on information in an event, a part editor might redraw its part, open or close windows, perform editing operations, transfer data, or perform any sort of menu command or other operation.

OpenDoc has several built-in event-handling features that help your part editor function properly within a compound document

2.2.2.1 The Document Shell and the Dispatcher

Part editors respond differently to user events than do conventional applications. Part editors do not directly receive events; OpenDoc receives them and dispatches them to the proper part.

Because they are not complete applications, and because they must function cooperatively, part editors run in an environment that itself handles some of the tasks that conventional applications typically perform. That environment is called the OpenDoc **document shell**.

Whenever an OpenDoc document is opened, the document creates an instance of the document shell. The shell opens the document window (or windows) and loads the part editors for all parts that appear in the document. The shell receives user events and uses the OpenDoc **dispatcher** to dispatch those events to the proper part editors, arbitrating ownership of shared resources such as menus when necessary.

2.2.2.2 Handling User Commands

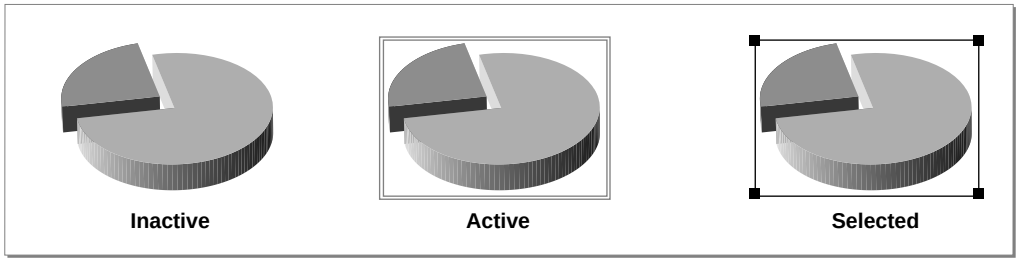
This section introduces how OpenDoc interacts with part editors to perform some common application activities that typically result from user actions or commands: part activation, menu-handling, data transfer, and Undo. In general, the document shell receives user commands, interprets them, and passes the appropriate information to the appropriate part.

2.2.2.2.1 Activation and Selection

Based on user actions, individual parts in a document become active and thus are capable of being edited. A part that is active is said to contain the current selection *focus*.

A different state from active is **selected,** meaning that the *frame* of the embedded part is made available for manipulation. Because embedded frames are considered to be content elements of their containing part, they can be selected and then moved, adjusted, cut, or copied just like text, graphic objects, or any other content elements. Thus, while an active part is manipulated by its own part editor, a selected part is manipulated, as a frame, by its containing part. Figure 12 shows the visual differences among the inactive, selected, and active states for an embedded part.

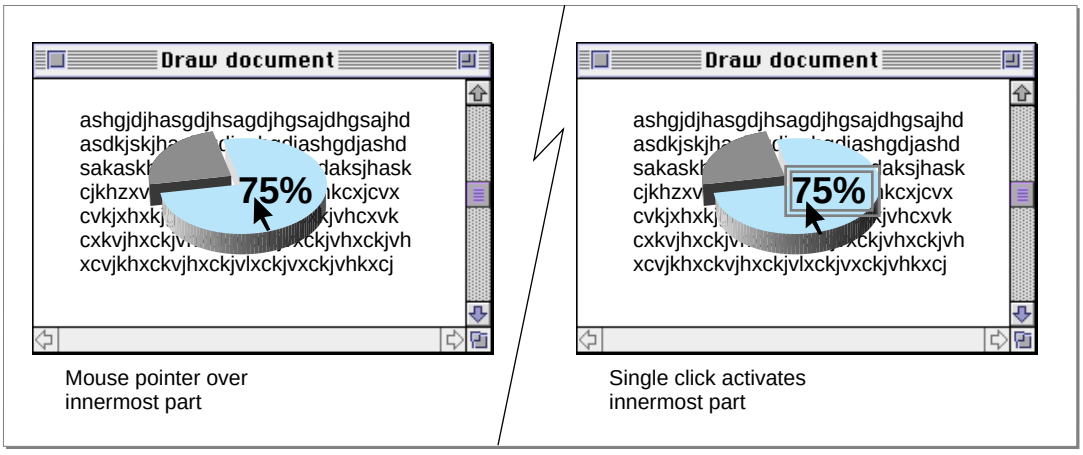
Figure 12 Inactive, active, and selected states of a graphics part



Note that an inactive part need not have a visible frame border. A selected part's frame border is drawn by the containing part; its shape typically corresponds to the frame shape. An active part's frame border is drawn by OpenDoc; its shape corresponds to the active shape of the embedded part's facet, and its appearance is fixed for each platform.

Activation generally occurs in response to mouse clicks within the area of a frame (specifically, within the area of an embedded facet's active shape). OpenDoc follows an **inside-out activation** model, in which a single mouse click causes activation of the smallest enclosing frame at the pointer location. In the case of a deeply nested embedded frame, as shown in Figure 13, this means that a single click within the frame of the most deeply embedded part activates that part, and allows the user to start editing immediately.

Figure 13 Inside-out activation of a deeply embedded part



By contrast, some compound-document architectures use an **outside-in activation** mode, in which the user selects the outermost (nonactive) frame with one click, and activates it with the second click. Thus, many clicks may be necessary for the user to activate a deeply embedded part. In Figure 13, for example, four clicks would be necessary for the user to start editing.

Despite the advantages of inside-out activation, there may be times when a containing part may not want an embedded part to be activated when the user clicks within its frame; it may instead want the part to become selected. For example, in situations where one wants the user to be able to move—but not edit—a part embedded within a part; selecting rather than activating by a mouse click. OpenDoc allows this behavior to be specified for an embedded part by placing its frame in a **bundled** state. A bundled frame acts like a frame with outside-in selection, in that a single click selects the frame. (However, unlike with outside-in selection, subsequent clicks do not then activate the part; it has to be unbundled before it can be activated.)

A part does not receive window-activation events directly. The document shell notifies a part when its window becomes active or inactive. It also notifies the containing part, rather than the embedded part, when an embedded part should become selected.

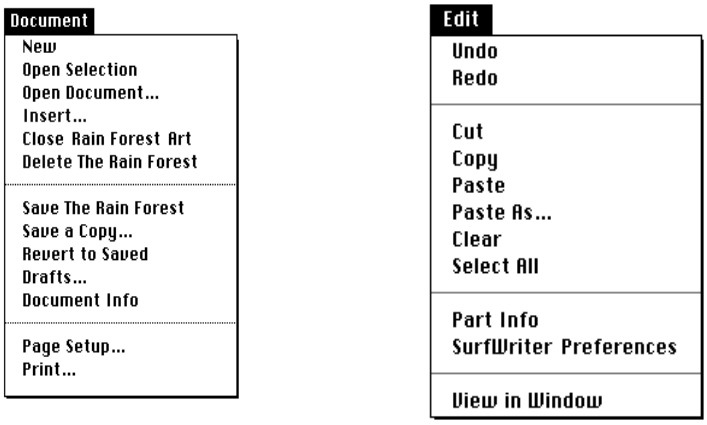
2.2.2.2.2 Menus

Menu organization and menu handling are different on different platforms. OpenDoc encapsulates certain aspects of menu behavior in a platform-neutral menu bar object. That object gives a part editor access to its own and to OpenDoc's menus, and allows t menu items to be converted to a standard set of command IDs.

The Document menu is basic to the OpenDoc user interface. The contents of the Document menu reflect the OpenDoc document-centered, rather than application-centered, approach. For example, there is no "Quit" command; it is replaced by a Close command. In OpenDoc, the user closes documents rather than quitting applications.

The Edit menu is another standard menu; it permits the user to operate on a part or its contents, when it is the active part. Figure 14 shows an example of the Document menu and the Edit menu for the Macintosh platform.

Figure 14 The Document and Edit menus



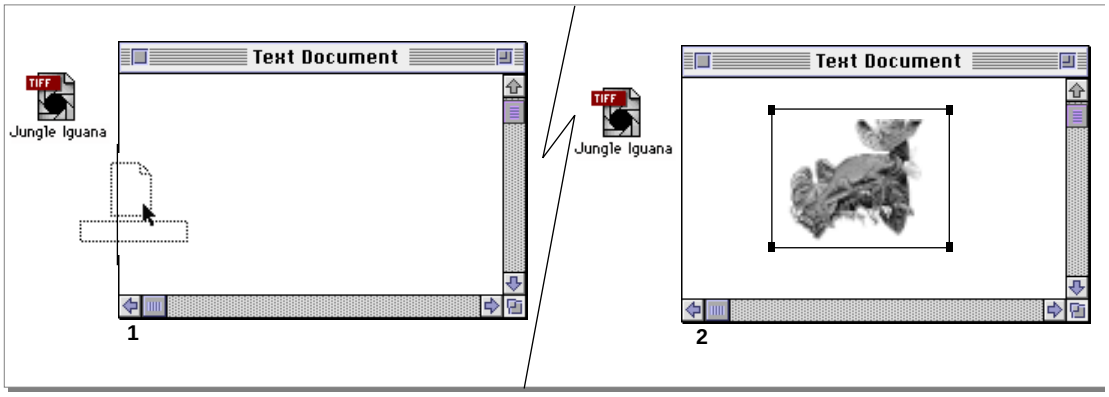
A part editor can add menus of its own, and it can modify the standard OpenDoc menus.

2.2.2.2.3 Data Transfer

Several built-in data-transfer features allow users to create and edit OpenDoc documents. Users can put any kind of media into a document with simple commands which OpenDoc transforms into a sequence of part editor method calls for execution:

- **Clipboard** data transfer allows for exchange of information among documents using menu commands. OpenDoc supports Clipboard transfer of any kind of data, including multipart compound data, into any document.
- **Drag-and-drop** data transfer is similar to Clipboard transfer, except that it involves direct user manipulation of the data being transferred, rather than the use of the Clipboard as an intermediary
- **Linking** allows the user to view, within a frame, part data that actually belongs to a different part in a different frame. Links in OpenDoc can be to parts in the same document or to parts in an entirely different document.

Figure 15 Using drag-and-drop from the desktop to a document



OpenDoc uses an intelligent form of pasting (or dropping) in data transfer, anticipating user expectation about the result of pasting operation when the destination holds a different kind of data from the source. Subject to user override, OpenDoc decides whether to embed the data as a separate part, or to translate it and incorporate it as content data into the destination part. Translation and embedding vs. incorporation are described under “Translation” on page 10.

Translation is minimized if developers export standard content formats.

2.2.2.2.4 Undo

Applications on many personal-computer platforms provide a form of **undo,** the ability of a user to reverse the effects of a recently executed command. Two aspects of OpenDoc undo are noteworthy:

- The undo action can cross document boundaries. This is important because a single drag-and-drop action can affect more than one part or document.
- OpenDoc allows multiple nested undo. The user can undo multiple sequential commands, rather than only one, if implemented by part editors.

2.3 Conceptual Model

2.3.1 The OpenDoc Classes

OpenDoc is a platform-independent, object-oriented programming interface. This section introduces the classes implemented in the OpenDoc libraries and the design goals behind them.

The interfaces to all of OpenDoc’s classes are specified in the **Interface Definition Language (IDL),** a programming-language-neutral syntax for creating interfaces

The IDL interface definitions are CORBA-compliant, meaning that they follow the language-neutral distributed dispatching standards of the Common Object Request Broker Architecture, established by the Object Management Group (OMG). Distributed dispatching is also supported in the basic design of OpenDoc:

- Almost all objects are instantiated by factory objects, rather than by constructors called by your part.
- OpenDoc minimizes the frequency of inter-part method calls, except in cases, like layout support, where frequent calls cannot be avoided.

For more information on the OpenDoc classes, including detailed descriptions of the platform-independent OpenDoc classes and methods, see the *OpenDoc Class Reference* in Part 2 of this submission.

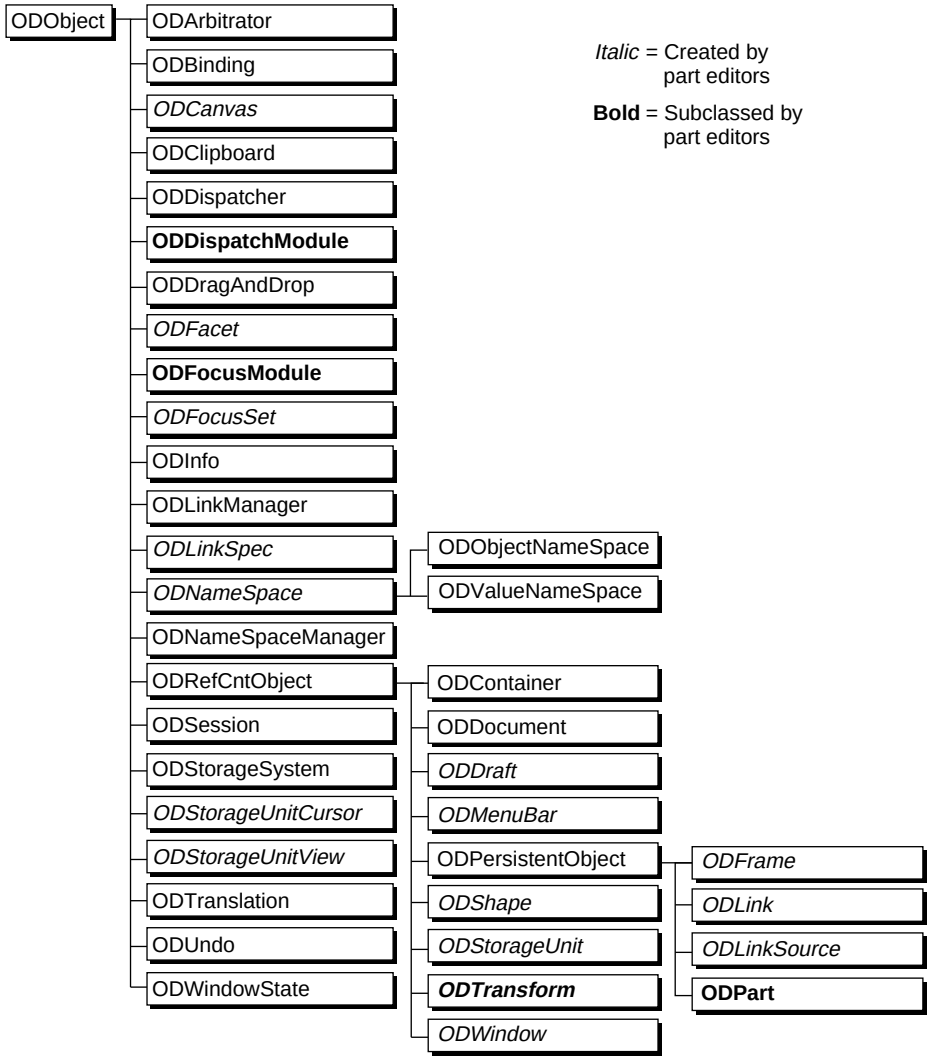
2.3.1.1 A Set of Classes, Not a Part-Editor Framework

OpenDoc is a set of classes whose objects cooperate in the creation and manipulation of compound documents. It is designed to be platform-neutral. The object-oriented structure, in which object fields are private, facilitates the replacement of existing code in a modular manner. Also, by using abstract classes and methods, it defines a structure for part editors while specifying as little as possible about their internal functioning.

OpenDoc classes are not an object-oriented application framework. That is, even though the design of OpenDoc enables one to create a part editor, it does so at a lower level. OpenDoc does not provide a functioning application that is modified via subclassing; it provides the facilities that enable the construction of part editors.

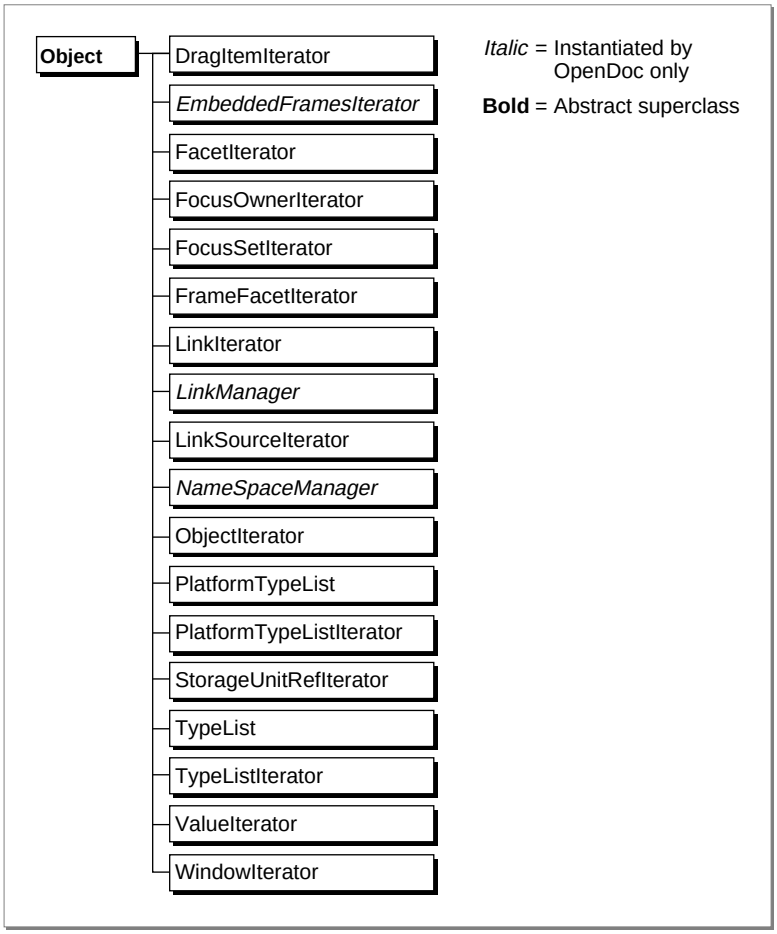
The principal classes in the OpenDoc class hierarchy is shown in Figure 16. Classes whose names are in bold are abstract superclasses; the other principal classes implement the basic capabilities of OpenDoc or represent extensions to its capabilities. Classes instantiated only by OpenDoc are in italics; other classes are typically instantiated by a part, if it uses them.

Figure 16 The OpenDoc class hierarchy (principal classes)



Classes shown in Figure 17 are support classes, consisting mostly of iterators and simple sets. They are all direct subclasses of `Object`.

Figure 17 OpenDoc class hierarchy (support classes)



Compared to some application frameworks, there is little inheritance in the hierarchy represented in Figure 16 and Figure 17; extensive use is instead made of object delegation, in which objects unrelated by inheritance cooperate to perform a task. This preserves the language-neutral flavor of OpenDoc and improves ease of maintenance and replaceability.

The OpenDoc classes can be divided into three groups, based on how a part editor might make use of them:

- The class (`Part`) that must be subclassed and implemented to create a part editor
- The existing OpenDoc classes, instantiated either by a part editor or by OpenDoc, that a part editor calls to perform its tasks
- The classes that can be subclassed and implemented to extend OpenDoc

These groups of classes are summarized in the following sections.

2.3.1.2 The Class Part

The class `Part` is the class that is subclassed to create a part editor. It represents the programming interface that a part editor presents to OpenDoc and to other parts.

`Part` is an abstract superclass with many defined methods. When `Part` is subclassed, all methods must be overridden, but meaningful implementations only need to be provided for those methods that represent capabilities supported by a part editor.

There is one additional class that must be subclassed and implemented if a part is a container part. An iterator class (a subclass of the abstract superclass `EmbeddedFramesIterator`) must be provided to allow callers to access all of the frames directly embedded in the part.

2.3.1.3 Other OpenDoc Classes

The OpenDoc classes listed in this section implement most of the OpenDoc features that a part editor uses. By using or instantiating objects of these classes, a part can function within an OpenDoc document and can embed other parts.

The OpenDoc classes themselves form several groups, based on whether they are abstract, whether they are implemented and support part-handling tasks, or whether they are service classes.

2.3.1.4 COSS

In order to take advantage of COS Life Cycle Services, the Object interface inherits from `LifeCycleObject`. Instances of Object and its children are created using `CosLifeCycle::GenericFactory`. The Key which causes an object of a given class to be created has an id equal to the name of the class, and a kind equal to “object interface”. The Criteria argument is not used.

GenericFactory creates an empty object, which should be initialized with `init_object`. For convenience, certain subclasses of Object have create operations for other subclasses; these should be implemented using calls to GenericFactory. They may also include object initialization. These operations include:

```
Arbitrator::create_owner_iterator
Arbitrator::create_focus_set
Document::create_draft
Draft::create_storage_unit
Draft::create_frame
Draft::create_part
Draft::create_link_spec
Facet::create_embedded_facet
Facet::create_facet_iterator
Facet::create_shape
Facet::create_transform
Facet::create_canvas
FocusModule::create_owner_iterator
FocusSet::create_iterator
Frame::create_facet_iterator
Frame::create_shape
Frame::create_transform
LinkManager::create_link
NameSpaceManager::create_name_space
ObjectNameSpace::create_iterator
Part::create_link
Part::create_embedded_frames_iterator
PlatformTypeList::create_platform_type_list_iterator
StorageUnitView::create_storage_unit_ref_iterator
Storage::create_container
Storage::create_type_list
Storage::create_platform_type_list
StorageUnit::create_view
StorageUnit::create_cursor_with_focus
StorageUnit::create_cursor
StorageUnit::create_storage_unit_ref_iterator
TypeList::create_type_list_iterator
```

```
ValueNameSpace::create_iterator
WindowState::create_window_iterator
WindowState::create_menu_bar
WindowState::create_canvas
WindowState::create_facet
```

Because `Object` is a Life Cycle Object, instances of `Object` or its subclasses should be deleted using the `remove` operation. For convenience, certain subclasses of `Object` have remove operations for other subclasses; these should be implemented using calls to the LifeCycleObject `remove` operation. These operations include:

```
Draft::remove_frame
Draft::remove_part
Draft::remove_link
Draft::remove_link_source
Facet::remove_facet
MenuBar::remove_menu
NameSpaceManager::delete_name_space
Part::remove_embedded_frame
PlatformTypeList::remove
Window::close_and_remove
```

`RefCountObject::release()` provides an implementation of `release()` which decrements an objects reference counter and removes the object when the reference count reaches zero.

The `copy` operation must be implemented for `MenuBar`, `Shape` and `Transform`. Note that some operations which might look like LifeCycle operations, such as `Dispatcher::remove_monitor` or `Facet::move_before` and `Facet::move_after`, are not LifeCycle operations.

`Object`t inherits from `IdentifiableObject` to provide the is_identical operator for object comparison. `IdentifiableObject` also provides the `ObjectIdentifier` attribute `constant_random_id`, which may be used to quickly determine that two objects are not equal to each other.

`NameSpace` inherits from `CosNaming::NamingContext` to allow COS Naming Services to be used.

2.3.1.4.1 Abstract Superclasses

These classes must be subclassed to be used:

- `Object`. This is the abstract superclass for most of the principal OpenDoc classes. All subclasses of `Object` define objects that are extensible.
- `RefCntObject`. This is the abstract superclass for reference-counted objects—objects that maintain a count of how many other objects refer to them, so that OpenDoc can manage memory use efficiently.
- `PersistentObject`. This is the abstract superclass for persistent objects—objects whose state can be stored persistently.

2.3.1.4.2 Implemented Classes

These classes are implemented in ways unique to each platform that supports OpenDoc:

- The session object. The class `Session` represents the user-editing session for a single OpenDoc document.
- The binding object. The class `Binding` represents the object that performs the binding of part editors to the parts in a document.
- Storage classes. The primary classes of objects associated with document storage are `StorageSystem`, `Container`, `Document`, `Draft`, and `StorageUnit`. The storage system object, container objects, document

objects, draft objects, and storage unit objects all work closely together and are implemented differently for each platform or file system.

- Data-interchange classes. The classes `Link`, `LinkSource`, `DragAndDrop`, and `Clipboard` all relate to transfer of data from one location to another. These objects do not represent documents, but they nevertheless use storage units to hold the data they transfer.
- Drawing-related classes. The classes `Canvas`, `Shape`, and `Transform` represent imaging structures in a given graphics system; the classes `WindowState` and `Window` represent windows on a given platform. The classes `Frame` and `Facet` represent the frame and facet structures that define the layout of embedded parts.
- Event-handling classes. The classes `Arbitrator` and `Dispatcher` control what kinds of user events are sent to which part editors during execution. The classes `MenuBar` and `Undo` give part editors access to the menu bar and to previous states of themselves, respectively.

2.3.1.4.3 Service Classes

A few object classes exist mainly as services for other classes to use:

- The `NameSpaceManager` and `NameSpace` classes provide convenient attribute/value pair storage. The classes `ObjectNameSpace` and `ValueNameSpace`, subclasses of `NameSpace`, provide, respectively, name spaces for OpenDoc objects and for pointers.
- The `Info` class provides the Part Info dialog box, a dialog box that your part displays as a result of a user selection from the Edit menu.
- The `Translation` class provides platform-specific translation between data formats.
- Many classes have associated iterator classes that are used for counting through all related instances of the class.

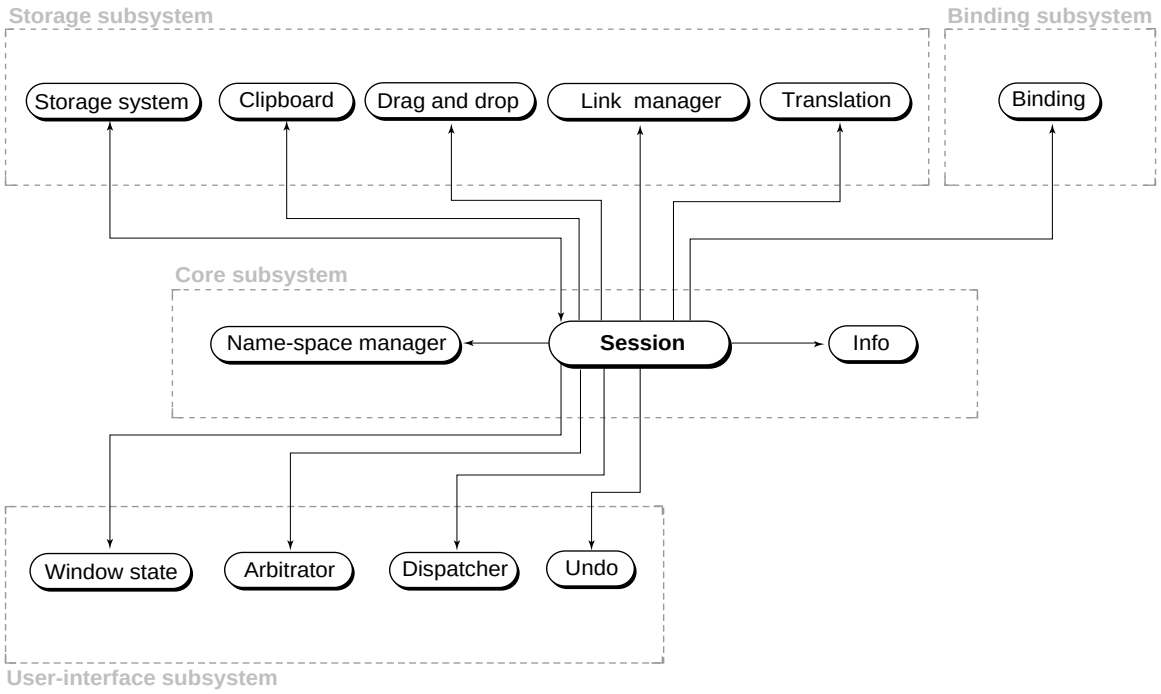
2.3.2 OpenDoc Object Relationships

This section gives a glimpse of the OpenDoc classes in action.The high-level run time state of an OpenDoc document involves relationships among a variety of objects instantiated from those classes. Taken together, the diagrams in this section show the principal run time relationships among the major OpenDoc objects for a single document. How those objects actually interact to perform their tasks is explained throughout the rest of this submission.

Objects cooperate in performing specific tasks within general categories, such binding, storage, and drawing. For example, Clipboard transfer is a specific task under storage, and frame negotiation is a specific task under drawing. The programming interfaces for performing those tasks are called **protocols,** and they are summarized in the section “Functional Overview” on page 33.

When an OpenDoc document is open, the session object occupies a central place in the relationships among the objects that constitute, support, and manipulate the document. Figure 18 shows the uppermost levels of that relationship.

Figure 18 Run time relationships of the session object



This figure, as do all subsequent object diagrams in this volume, uses the following conventions to describe run time relationships:

- Individual objects are labeled boxes.
- Arrows between the boxes show object references (equivalent to pointers in C++), as defined primarily by the availability of accessor functions for them in the public interface to OpenDoc. The references as shown here may not strictly mirror the actual (private) references that exist between OpenDoc objects.
- A mutual reference between a pair of objects is represented by an arrow with a head on each end.
- A one-to-many relationship, in which one object can simultaneously reference more than one other object of a given kind, is represented by an arrow with a double head on one end.
- As Figure 18 shows, the session object maintains many (mostly one-way) relationships with other objects. That means a part gains access to many objects through the session object. The objects are grouped into categories based on the kinds of tasks they perform together; the following subsections discuss and expand upon the object relationships in terms of those categories. Note also that Figure 18 is incomplete; Figure 19 on page 26 and Figure 22 on page 28 continue the object-reference hierarchy.

2.3.2.1 Binding

OpenDoc uses the **binding object,** directly referenced by the session object, to pick the proper part editor for each part in a document, both when the document is opened and whenever a new part is added to it.

The binding object makes use of the COSS naming service.

If no editor is available that can manipulate the data of a part, but if platform-specific translation facilities are available, the **translation** object, a wrapper for the data translation services, can be used by OpenDoc (as directed by the user) to translate the data of a part into a part kind for which an editor is available.

2.3.2.2 Drawing

The objects that make up the OpenDoc drawing capability provide a set of platform-independent protocols for embedding (placing embedded frames within the content of a part), layout (manipulating the sizes and locations of embedded frames), and imaging (making part content and frames visible in a window). They describe part geometry

in a document, whether for printing or for screen display. They permit manipulation of windows, frames, and clip shapes, and provide wrappers for certain platform-specific imaging structures.

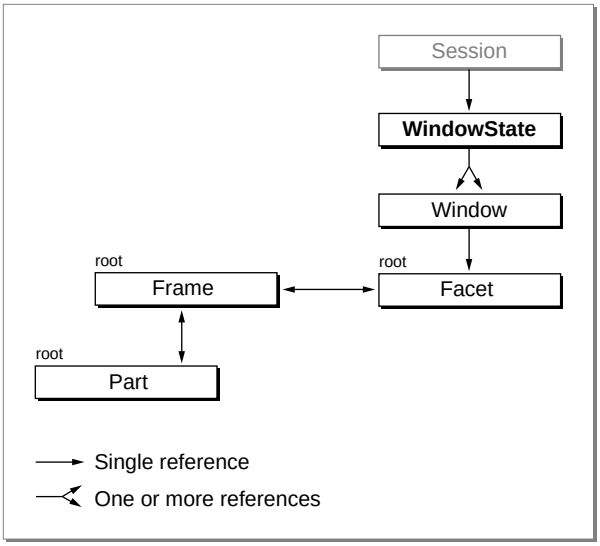
The drawing capability of OpenDoc is not an imaging model. A part editor must make platform-specific calls to actually draw the lines, polygons, text, and so on, that make up the part.

The object relationships involved in drawing can be addressed through the window state object.

2.3.2.2.1 The Window State and Windows

Figure 19 is a simplification of the run time object relationships among the objects involved with a document window. It is a continuation of one branch of Figure 18 on page 25, and shows the upper-level relationships among windows and the parts they contain.

Figure 19 Window-related object relationships



As Figure 19 shows, the **window state** object is referenced by the session object. It is a list of all the open windows in which OpenDoc parts are displayed. (Note from Figure 18 on page 25 that the session object also references the menubar object; see “Interface Elements” on page 30 for more information.)

The window state object references one or more **window** objects, which are wrappers for platform-specific window structures.

Each window object references a single **facet** object (the *root facet*), the visible representation of the **frame** object (the *root frame*) of the outermost part (the *root part*) in the document being displayed in the window. The root frame in turn references the root part, which controls some of the basic behavior (such as printing) of the document. The root frame and root facet fill the content region of the window.

This relationship of window to root facet, root frame, and root part applies regardless of whether the window is a document window or a part window opened up from an embedded frame.

The embedding relationships among facets, frames, and parts shown in Figure 19 is not complete; it continues down through all parts in the document. The root facet references its embedded facets, and the root part references its embedded frames, as shown in the next section.

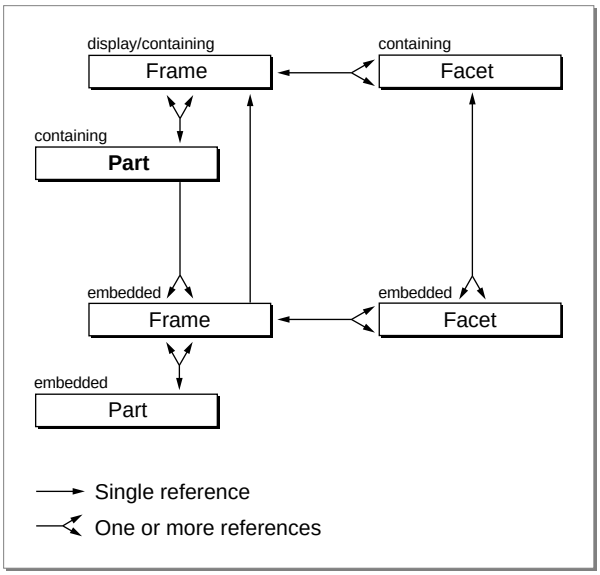
2.3.2.2.2 Embedding

Figure 20 is a picture of the relationships among facet, frame, and part objects at any level of embedding in a document. It can be thought of as a direct continuation of Figure 19, but it applies to deeper levels as well.

The first thing to note from Figure 20 is that embedding is represented by two separate but basically parallel structures. On the left, each part references one or more display frames above it (the frames within which its contents are displayed), and one or more embedded frames below it. Those embedded frames, in turn, reference the parts for which they are the display frames. The document embedding structure, then, is represented by a frame-to-part, frame-to-part sequence in which each part only indirectly references its embedded parts. Furthermore, note also that an

embedded frame does not directly reference its containing part; instead, it references a display frame of its containing part.

Figure 20 General embedding object relationships



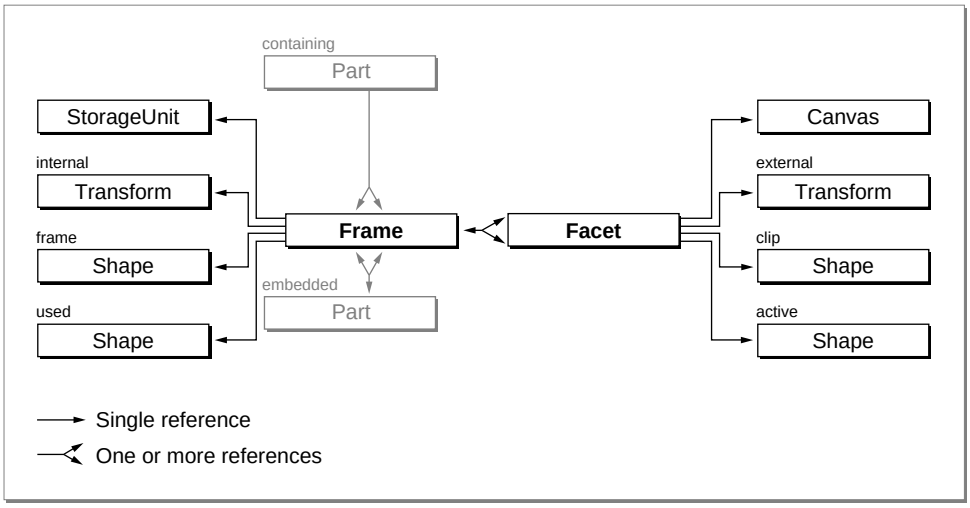
On the right side of Figure 20, the facets for the frames form their own, simpler hierarchy. Each containing facet directly references its embedded facets, and each embedded facet directly references its containing facet. This more direct imaging hierarchy allows for fast event dispatching by OpenDoc.

Connecting the two hierarchies are the frame-to-facet references. Each frame that is visible references the facet or facets that correspond to it. (A frame can have more than one facet.) Each facet likewise references its frame. Note that facets need not exist unless their frames need to be drawn.

2.3.2.2.3 Layout and Imaging

The frame and facet objects shown in Figure 20 have additional references besides those shown. A part's display frame (or frames) and facet (or facets) use several other OpenDoc objects when laying themselves out and preparing to draw the content of their part. Figure 21 shows these additional relationships, for a given frame-facet pair at any level of embedding.

Figure 21 Layout and imaging object relationships



Each display frame object for the part being drawn includes references to these objects:

- a transform object (the **internal transform**) that defines the geometric offset or transformation of the part within its frame
- a shape object (the **frame shape**) that defines the shape of the frame
- another shape object (the **used shape**) that defines the portion of the frame shape that is actually drawn into
- a storage unit, for storing the state of the frame into its document (unless it's a *nonpersistent frame*)

Each of the facet objects for the part being drawn represents an area within a window (or printer image) that corresponds to a visible display frame (or part of a frame) of the part. The facet includes references to these other objects that hold platform-specific drawing information:

- a **canvas** object that describes a platform-specific drawing context or structure
- a **transform** object (the **external transform**) that defines an external geometric offset or transformation for the facet within the containing part
- a **shape** object that defines the **clip shape** for the facet (what portion of the frame shape get drawn)
- another shape object that defines the **active shape** for the facet (what portion of the frame shape accept events)

During the layout and imaging process, a part editor is typically asked to draw the contents of a particular facet. The part editor gets the clipping, transformation, and layout information from the facet and its frame, and then makes platform-specific graphics calls to perform the actual drawing.

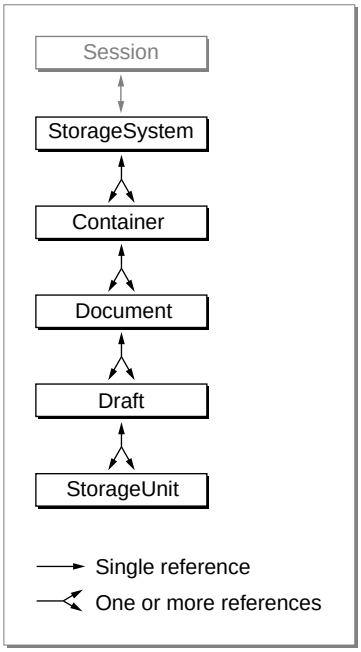
2.3.2.3 Storage

The storage capabilities of OpenDoc include both document storage and data transfer. As Figure 18 on page 25 shows, three objects directly referenced by the session object are involved with storage issues.

2.3.2.3.1 Document Storage

- OpenDoc manages persistent storage for parts and other objects in documents. Storage in OpenDoc consists of structured storage elements that can contain many data streams. Figure 22 shows the main storage-related objects and their run time relationships. It is a continuation of one branch of Figure 18 on page 25.

Figure 22 Document-object relationships



These objects combine to make up a **container suite,** a specific implementation of the OpenDoc storage architecture. Container suites can be implemented in different ways on different platforms. Here is how the objects relate to each other:

- The session object instantiates the storage system object, and the **storage system** object instantiates and maintains a list of **container** objects.
- Each container may contain one (or possibly more, depending on the capabilities of the container suite) **document** objects, each of which represents an OpenDoc document.
- Each document contains one or more **draft** objects. Each draft is a named configuration of its document that represents a snapshot of the document's state at a particular moment.
- Each draft contains a number of **storage unit** objects. Each storage unit can contain several different data streams, all of which provide information about the object to which the storage unit applies. The data streams in a storage unit are identified by **property,** the kind of information contained, and **value,** the data type of that information. A storage unit can hold more than one property, and a property can hold streams of more than one value.

The object whose data is stored in a storage unit also references the storage unit. As Figure 23 on page 30 shows, for example, the part whose data is stored in a draft has a reference to its main storage unit, which in turn can reference other storage units.

2.3.2.3.2 Data Transfer

OpenDoc storage is used for other kinds of data manipulation besides document storage. Clipboard transfer, drag-and-drop, and linking all make use of storage unit objects.

Figure 18 on page 25 shows the following objects, directly referenced by the session object, that are used in data transfer:

- The Clipboard object, instantiated at the initialization of the session object, references one or more storage units that represent the data held on the Clipboard.
- The drag-and-drop object is instantiated when the session object is initialized. Like the Clipboard, it references one or more storage units that represent the data to be transferred by the drag.

Figure 23 shows additional object references maintained by the part object, beyond those shown in Figure 20 on page 27, that facilitate data transfer.

- A part object that contains the source of a link references a link source object, instantiated by the part that contains the source data. The link source references a storage unit that holds a copy of the source data for the link, and one or more link objects.
- A part object that contains the destination of a link references a link object. The link object references a storage unit that holds a copy of the source data for the link. (Links and link sources for an intra-document link share the same storage unit.)

Figure 23 Part-storage relationships

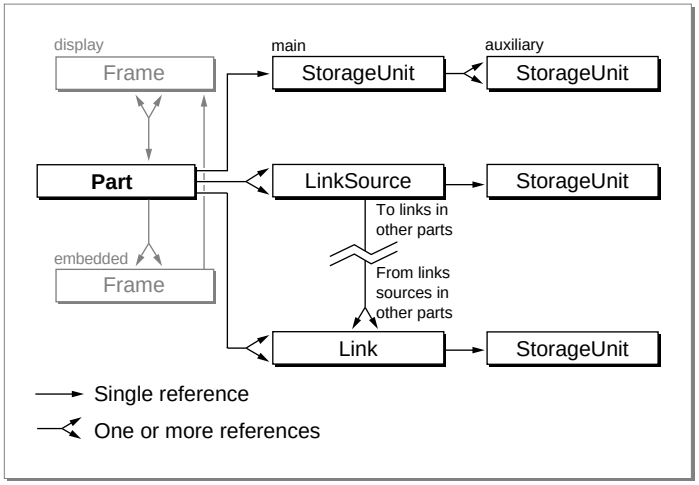


Figure 23 also shows how a part organizes its document storage. To store its data in its draft, the part has a reference to its one **main storage unit,** which in turn can reference other **auxiliary storage units.**

2.3.2.4 User Interface

The objects that handle user interaction with the parts in a document are mainly concerned, as Figure 18 on page 25 shows, with event dispatching and with handling user commands and constructing user interface elements.

2.3.2.4.1 Event Dispatching

User events are handled by two primary objects referenced by the session object, the dispatcher and the arbitrator. Together, they deliver events to the appropriate part editor.

- The **dispatcher** accepts user events from the underlying operating system through the document shell, and dispatches them to part editors. The dispatching system is modular and can be extended to handle new classes of events by adding **dispatch module** objects. Each dispatch module is responsible for deciding which frame or part should be asked to handle each event that the module deals with.
- The **arbitrator** negotiates temporary ownership of a **focus,** a shared resource such as the menu bar, keystroke stream, and so on. Like the dispatcher, the arbitrator is modular. Through addition of **focus modules,** it can be extended to handle new classes of software or hardware resources as defined by part editors or individual platforms.

2.3.2.4.2 Interface Elements

Other objects referenced directly or indirectly by the session object facilitate the handling of user commands, or are involved with constructing user-interface elements.

- The **undo** object, an object that stores command history for all parts in the document, is referenced by the session object. It allows parts to reverse or restore the effects of multiple previous user commands. There is a single, session-wide undo object used by all parts in a document.
- The **menu bar** object, referenced by the window state object, represents the base menu bar created by OpenDoc. A part copies and adds to that menu bar to create its menus.

3. Specification

3.1 IDL Description

Volume II of this submission, the Class Reference, is organized alphabetically. This makes it easy to locate a class from its name, but makes it difficult to determine which classes comprise which facilities. In order to make this task easier, the following subsections list the classes in each facility.

3.1.1 Compound Presentation Facility:

See “Appendix C - IDL for the Compound Presentation Facility” for the class definitions. See “Appendix F - Types and Constants” for the related data type and constant definitions.

Arbitrator
Canvas
Dispatcher
DispatchModule
Facet
FacetIterator
FocusModule
FocusOwnerIterator
FocusSet
Frame
Info
MenuBar
ObjectIterator
Shape
Transform
Window
WindowState

3.1.2 Compound Interchange Facility:

See “Appendix B - IDL for the Compound Interchange Facility” for the class definitions. See “Appendix F - Types and Constants” for the related data type and constant definitions.

Clipboard
Container
Document
Draft
DragAndDrop
Link
LinkManager
LinkSource
LinkSpec
StorageSystem
StorageUnit
StorageUnitCursor
StorageUnitView

3.1.3 Compound Core Facility:

See “Appendix D - IDL for the Compound Core Facility” for the class definitions. See “Appendix F - Types and Constants” for the related data type and constant definitions.

Binding
Object

Part
PartWrapper
PersistentObject
RefCntObject
Session
Translation
Undo

3.1.4 Compound Utilities Facility:

See “Appendix E - IDL for the Compound Utilities Facility” for the class definitions. See “Appendix F - Types and Constants” for the related data type and constant definitions.

DragItemIterator
EmbeddedFramesIterator
FocusOwnerIterator
FocusSetIterator
FrameFacetIterator
NameSpace
NameSpaceManager
ObjectNameSpace
PlatformTypeList
PlatformTypeListIterator
StorageUnitRefIterator
TypeList
TypeListIterator
ValueIterator
ValueNameSpace
WindowIterator

3.2 Behavioral description

Full behavioral descriptions of the classes are given in Volume II of this submission, the Class Reference. Classes are listed in alphabetical order. Each class definition includes a full description followed by a discussion of the methods for the class. A quick summary of the methods is given, along with the logical grouping in which the methods are presented. A discussion of each method follows that includes a description of the parameters, return values, and exceptions.

4. Functional Overview

OpenDoc is an architecture that facilitates the creation, manipulation, and interchange of compound documents. This architecture is realized as a set of classes and their associated methods. This section presents an overview of the OpenDoc architecture from the part editor developer's point of view. This section illustrates the relationship between the classes and methods, and provides a roadmap that guides the reader to the detailed descriptions of the classes and methods given in the Class Reference volume of this submission.

Note that the Class Reference is the final authority on OpenDoc; many details are omitted in this section in order to present the larger view.

4.1 Functional Class Structure

Although the class hierarchy diagrams given in Figure 16 and Figure 17 are useful for showing inheritance, an alternate functional grouping assists in understanding the use of the classes. There are three functional groupings of classes and methods:

- those used by part editors,
- those used by document shells, and
- support services.

Be aware that there is some overlap between these groups.

All classes and methods are available to a part editor developer; there is no protection that prevents a part editor from calling a document shell method, just as there are no protections against other programming errors. The distinction between part editor methods and document shell methods is clearly made in the Class Reference volume.

4.2 Fundamental Concepts

The following subsections describe some general concepts that are used throughout OpenDoc.

4.2.1 What Is OpenDoc?

OpenDoc provides an environment for part editors, as discussed in the "Architectural Model" section. There are two elements that make up this environment. The first is the set of classes and their methods that are defined in the Class Reference volume. The second is the **document shell**.

The classes function in concert with the document shell. The document shell is the glue that binds the classes together into an executable program. The document shell encapsulates most of the OpenDoc system dependencies, which keeps the classes more portable. These system dependencies include the much of the user interface, and the mechanisms used to instantiate and communicate with part editors.

4.2.2 Bento

Several methods take "Bento" parameters. Bento is an implementation of the storage system used in OpenDoc.

4.2.3 Pointing Devices

OpenDoc expects the existence of a two-dimensional locator input device on the target platform. It is also expected that such an input device has one or more buttons associated with it. The term **mouse** is used to refer to such devices in this document.

4.2.4 Exceptions

OpenDoc is designed for an asynchronous, multi-process, distributed environment. As a result, exceptions are not always generated immediately. For example, a method call to a clipboard may not fail immediately; it may take some time for the clipboard to report an error back, and this may be after the original method call has completed. Therefore, an exception may appear in response to a seemingly unrelated subsequent method call.

4.2.5 Lifetime of Returned Values

Part code is called by OpenDoc. This code, in turn, may call other OpenDoc methods. In many cases, the values returned by these methods become invalid when the part code returns control to OpenDoc. These cases are discussed in the Class Reference.

4.2.6 Factory Methods

Any OpenDoc object that a part editor instantiates must be created by a **factory method,** a method of one class used to create an instance of another class. Undefined behavior will result if target programming language object instantiation mechanism are used, such as using the C++ `new` operator. The **Description** section of each class in the Class Reference lists the factory methods used to instantiate objects of that class.

4.2.7 Reference-Counted Objects

OpenDoc uses reference-counted objects as part of its internal memory management scheme. **Reference-counted objects** are objects that maintain a count of the current number of references to them; that is, they are shared objects that are aware of how many other objects are making use of them at any one time.

Reference-counted objects do not transparently know how many other objects are making use of them; their reference counts are explicitly incremented or decremented by method calls that act on those objects. The **Discussion** section of a method in the Class Reference indicates how the method affects reference counts.

A reference count is a non-negative integer. A value greater than 0 means that at least one reference to the object currently exists, and thus the object must not be removed from memory. All objects of the class `RefCntObject` and any of its subclasses (including frames, links, link sources, and parts) are reference-counted. The `Object` class, which is a superclass of the `RefCntObject` class is implemented using the COSS Life Cycle Services. The `Object` class is a subclass of the `CosLifeCycle::LifeCycleObject`.

Each reference-counted object is created by a call to its factory method. When it is first created by its factory object, a reference-counted object has a reference count of 1. For example, the frame object returned by a `Draft` object's `create_frame` method always has a reference count of 1, because that method always creates a new object. The frame object returned by a `Draft` object's `get_frame` method, however, may have a reference count greater than 1, because that method may return a new reference to a pre-existing frame.

Calling a method that decrements the reference count of a reference-counted object when its reference count is 1 causes its reference count to be set to 0. Reference-counted objects become invalid when their reference counts reach 0; it is an error to use such an object in any way including incrementing or decrementing its reference count because something else may be stored in the memory that held the object.

The reference count of each reference-counted is available through the `get_ref_count` method. Objects are responsible for notifying their factory objects when their reference counts become 0.

It is essential that reference-counted objects always be released using the appropriate method call rather than using some system or language specific deletion technique. Any method call that returns a reference-counted object must be balanced by a subsequent `release` call, for example:

```
clip_shape = facet->acquire_clip_shape(ev, bias_canvas);
...
clip_shape->release(ev);
```

Some methods automatically increase the reference counts of their reference-counted object parameters. See the Class Reference to determine which methods affect reference counts. In these cases, the reference count must be explicitly decremented if the caller does not need the object any longer, for example:

```
new_shape = facet->create_shape(ev);
...
facet->change_clip_shape(ev, new_shape, bias_canvas);
new_shape->release(ev);
```

4.2.8 Iterators

Many OpenDoc objects maintain lists. These objects have associated iterator objects that are used to provide access to all of the items in the lists. For example, the `FrameFacetIterator` object is used to access all of the elements in a `Frame` object's facet list.

Most of the iterators have three common methods. The `first` method returns the first item in the list, the `next` method returns the next item in the list, and the `is_not_complete` method tests for the end of the list.

4.2.9 Tokenized Strings

OpenDoc uses ISO character strings to describe many things by name. These strings tend to be quite long, and therefore it is inefficient to perform frequent operations on them. OpenDoc maintains a registry that makes such strings into integer tokens for efficient use. The `Session::tokenize` method registers a string and returns a token.

4.2.10 Properties

OpenDoc includes the concept of properties, which are named units of storage. There are a number of standard properties defined in the Class Reference volume.

The most important property is `prop_contents`. Every part (or other persistent object) has a single **main storage unit,** whose `prop_contents` property stores the content for that object. Not all of the part's contents need actually be stored in that storage unit however; **auxiliary storage units** can be created that hold additional data related to the data in a main storage unit as long as the main storage unit contains strong persistent references to the auxiliary storage units.

4.2.11 Focus

OpenDoc uses the concept of a focus to manage access to shared resources by parts. A `FocusModule` object exists for each such resource. A collection of `FocusModule` objects may be managed by a `FocusSet` object which allows them to be negotiated for as a group. The `Arbitrator` object handles the actual arbitration of focus modules and focus sets.

A part is said to own the focus for a resource or collection of resources if it has been granted access to it or them.

4.2.12 Window Management and Drawing

OpenDoc does not define any mechanism for drawing; part editors use platform-specific drawing packages. OpenDoc does, however, define mechanisms for the arbitration of areas for drawing into.

Parts have a hierarchical relationship with each other, as discussed in 2.2.1.1.2, "Frames and Embedded Parts". A side-effect of this relationship is that visible portions of some parts obscure portions of other parts, thus giving rise to OpenDoc's window management facilities.

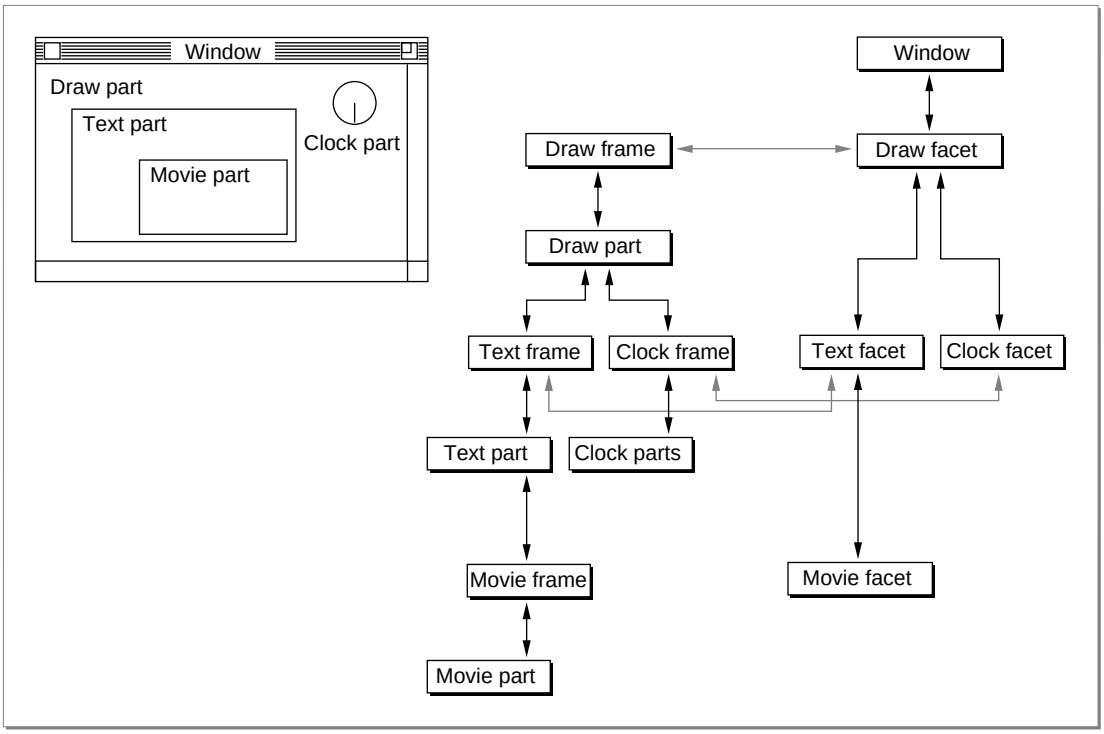
The `Session` object creates a `WindowState` object which manages all windows which are managed by `Window` objects. `Window` objects are wrappers that map external platform-specific windows into internal OpenDoc windows. `Frame` objects are associated with `Window` objects, and describe the display area for a part editor in the window. `Facet` objects are associated with `Frame` objects, and represent visible areas of a `Frame` that can be drawn into. `Canvas` objects are associated with `Facet` objects, and provide access to platform-specific information that a part uses for drawing using a platform-specific drawing package.

Each window has a root frame. Embedded frames can be created by the containing part. An embedded frame has a containing frame which is the containing part's frame inside which it is embedded.

The object hierarchy of embedding determines how information is passed among the parts that make up a compound document. There are effectively two hierarchies. Parts and embedded frames make up one hierarchy; facets make up a separate but essentially parallel hierarchy.

Figure 24 shows a simple example of these hierarchies for a document that consists of a graphics root part ("draw part"), in which is embedded a clock ("clock part") and a text-processing part ('text part"). The text part has a movie-viewing part ("movie part") embedded within it. As in the diagrams shown in 2.3.2, "OpenDoc Object Relationships", the objects are represented by boxes, with the most important references between them drawn as arrows.

Figure 24 Simplified frame and facet hierarchies in a document



The window object is at the top of Figure 24, uniting the two hierarchies. The window object itself is referenced by the window state object, as shown in Figure 19 on page 26. Some interobject references have been omitted for simplicity.

The fundamental structure of embedding in the compound document is represented by the hierarchy on the left. The draw part is the root part of the document; it directly references its two embedded frames. Those frames in turn reference their parts, the text part and the clock part. The text part references its one embedded frame, which in turn references its part, the movie part. Each part thus only indirectly references its own embedded parts. (In this case, there is one frame per part, but there could be more than one.)

Note also that each part also has a reference back up the hierarchy, to its display frame. Not shown in this figure is the reference from each frame directly to its containing frame. These references were omitted for the sake of clarity. See Figure 19 on page 26 for a more complete picture of the object structure of embedding.

The embedding relation of parts and frames does not have to be the strict hierarchy shown in Figure 24. Parts can have frames that contain frames.

A Parts' content and frames are stored in their documents. When the window is closed, the states of all the parts and frames are saved. When the window is reopened, all the parts and frames are restored by reading in the saved data.

The hierarchy on the right side of Figure 24 is analogous to the one on the left side, but it is simpler and more direct. The facet hierarchy is designed for drawing and event-dispatching. Each facet corresponds to its equivalent frame, but it directly references its embedded facets. (In this case, there is one facet per visible frame, but there could be more than one.)

Whereas frames must exist for all embedded parts in a document, facets are needed for only those frames that are visible at any one moment. Since the facet hierarchy is for drawing and event-dispatching, there is no need to have facets that can't be drawn and can't accept events.

Facets are ephemeral; they are not stored when a document is saved. Facets are created and deleted by their frames' containing parts, as the facets' frames become visible or invisible through scrolling, frame or window resizing, or window opening and closing.

Note from Figure 24 that each frame has a direct reference to (and from) its equivalent facet. That frame-to-facet reference is the connection between the two hierarchies. It also means that each part object references its own facets only indirectly, through its display frames.

4.2.13 Windows and Canvases

Windows and canvases are inherently platform-specific. The OpenDoc objects used to represent them are basically wrappers for structures that differ across platforms.

4.2.13.1 Windows

The OpenDoc class `Window` is a wrapper for a pointer to a platform- specific window structure. For some operations, a part editor must retrieve the window pointer from the `Window` object, and use the platform's facilities. In most cases, however, the interface to `Window` provides the capability needed for interacting with windows.

4.2.13.1.1 The Window State Object

There is a single instantiated `WindowState` object for each OpenDoc session. The window state consists of a list of the currently existing window objects. The access to all windows, and hence all facets, frames, and parts, is through the window state.

The document shell and arbitrator use the window state to pass events to parts so that they can activate themselves, handle user input, and adjust their menus as necessary. A part may be displayed in any number of frames, in any window of a document. The arbitrator uses the window state to make sure that it passes events to the correct part, no matter what window encloses the active frame and how many other frames the part has.

Normally, a part editor calls the window state only when it creates new windows and when it needs access to a particular window.

If for some reason a part needs access to all windows, it can create an `WindowIterator` object, which gives it access to all windows referenced by the window state.

4.2.13.1.2 Creating and Registering a Window

Windows in OpenDoc are maintained through the `WindowState` object, which is obtained from the session object. After a window is created with platform-specific calls; the `register_window` method of `WindowState` is then called to create an OpenDoc window object describing the platform-specific window.

A window has an *is-root* property. A **root window,** which is the same as a **document window,** has a root part that is the root part of its document. If a window's is-root property is not set, it is a **part window** that has been opened from a **source frame** within a root window. The document associated with a document window is kept open as long as the document window is open. (OpenDoc permits multiple document windows for a single document, as long as the root part provides a user interface to support it.) The document shell closes a document when its last document window is closed.

Windows also have a *should-save* property that, if set, means that the state of the window is saved persistently after the window closes. Usually, only root windows are marked as should-save.

The creator of a window can specify the view type and presentation of the **root frame**, the frame that displays the root part. The view type specifies whether or not the root part should draw itself as an icon, and the presentation specifies what kind of appearance the part content should have if not drawn as an icon. View type and presentation are suggestions to the part editor that draws within that frame.

OpenDoc assumes that each window has a single canvas, attached to its **root facet,** the facet created for the root frame. On the Macintosh, for example, the root frame in the window has the same shape as the window's content region: it includes window scroll bars but excludes the window's resize box, if present.

A window should be created as invisible, and then made visible as described next.

4.2.13.1.3 Opening a Window

After creating a window, a part editor typically makes calls to these three methods, in this order:

1. the window's `open` method, which creates the root frame
2. the window's `show` method, which adds a facet to the root frame and makes the window visible

3. the window's `select` method, which brings the window to the front

4.2.13.1.4 Window IDs

A part editor does not maintain references to `Window` objects for accessing OpenDoc windows, because the document shell or the window state object can close a window and invalidate the reference. Instead, the window state assigns window IDs that are valid for the length of a session. Use the window's `get_id` method to get the ID of a window when it is created, and then pass that ID to the window state's `get_window` method for subsequent access to the window.

4.2.13.1.5 Closing a Window

If a part editor needs to close a window programmatically, it should call the window's `close_and_remove` method. That closes the window, releases the window object, deletes the root facet, and removes the root frame from the document. It also makes any necessary platform-specific calls to dispose of the window itself.

4.2.13.1.6 The Open Method of a Part Editor

Opening a part means creating a window for it and displaying it in the window. A window has to be opened only if the part is to be the root part of that window. The part's `open` method is called, therefore, in these circumstances:

- when the part is the root part of a document being opened
- when the part is initially created from stationery (it comes up in a document window if created from a stationery document, and in a part window if created from an embedded stationery part)
- when the user chooses the View In Window command to open the embedded part's (active) frame into a part window
- when the user chooses the Open command to open the embedded part's (selected) frame into a part window

When a document window is opened, a null value for the `frame` parameter is passed to the `open` method; when a part window is opened, a reference to the frame whose contents are to be viewed in the window is passed to the `open` method.

A part's implementation of `open`, can take steps similar to this:

- If opening a frame into a window:
    - If the window already exists, get a reference to the window by calling the window state's `get_window` method, and then bring the window to the front by calling its `select` method.
    - If the window does not yet exist, call the part's private method to create a platform-specific window and then register it with the window-state object. Next, get the window's ID (from its `get_id` method). Finally, call the window's `open`, `show`, and `select` methods to make it visible.
- If opening a document window, call the part's private window-creation method and then register the window. Get and store the window's ID. Call the window's `open`, `show`, and `select` methods to make it visible.

4.2.13.2 Using Canvases

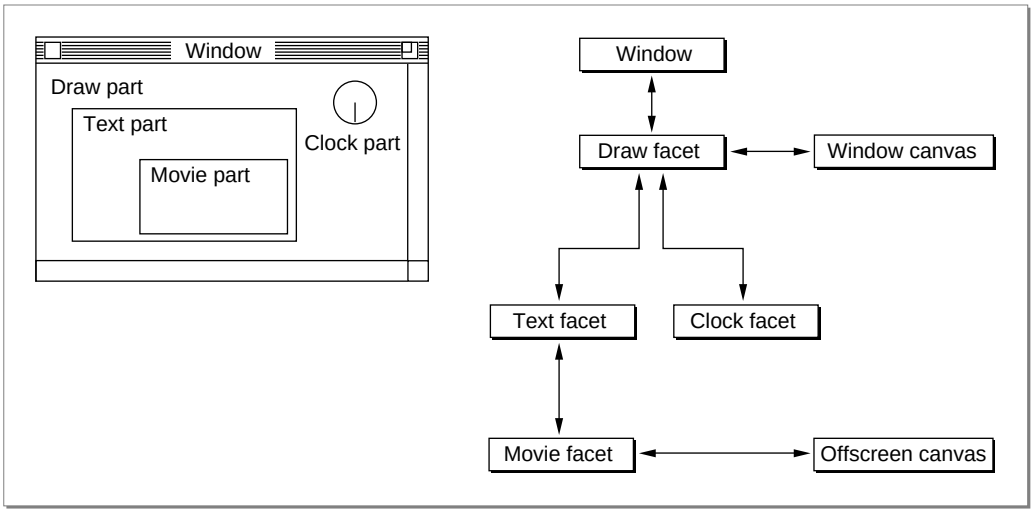
The class `Canvas` is OpenDoc's wrapper for a platform-specific (or graphics-system-specific) structure that represents a drawing environment. A drawing canvas can refer to anything from a bitmap or a structured display list to a stream of PostScript code. It represents the destination for drawing commands, the environment for constructing a rendered image; it has a coordinate system and may retain state information (such as, for example, pen color) that influences how drawing commands are interpreted.

Canvases can be dynamic or static, meaning that they are used for video display or printing display, respectively. A part editor can determine whether it is drawing to a static or dynamic canvas, and adjust its procedures as necessary.

Canvases can also be onscreen or offscreen

Canvases are attached to individual facets. In Figure 25, for example, which shows the same document with the same facet hierarchy as in Figure 24 on page 36, the document has two attached canvases. An offscreen canvas is attached to the movie part's facet; its **parent canvas** (the one immediately above it in the facet hierarchy) is attached to the root part's facet. (A canvas attached to the root facet of a window is also called a **window canvas;** every window has a window canvas, assigned to its root facet by the window-state object when the part first registers the window.)

Figure 25 Facets and canvases



If a particular facet in a window's facet hierarchy has an attached canvas, all of its embedded facets (and their embedded facets, and so on) draw to that canvas. Thus, for most drawing, only a window's root facet needs a canvas. In Figure 25, for example, any drawing done to the text facet, draw facet, or clock facet ends up being rendered on the window canvas.

However, if a particular part needs an offscreen canvas, for itself or for an embedded part, it can attach a canvas to a facet anywhere within the hierarchy. Any drawing done to the movie facet in Figure 25, for example, is rendered on the offscreen canvas.

Every canvas has an **owner,** a part that is responsible for transferring images drawn on its canvas to the canvas's parent canvas. in Figure 25, for example, the movie images drawn to the offscreen canvas must be transferred to the window canvas in order to be viewed onscreen. The owner decides when and how to transfer images from a canvas to its parent. The owner of the offscreen canvas in Figure 25 might be the movie part, or the text part that contains the movie part, or another part.

4.2.14 Transformations

OpenDoc deals with a multitude of two-dimensional coordinate systems. These include the coordinate system for the platform display, printer coordinate systems, and the coordinate systems for the windows, frames, facets, and canvasses. OpenDoc transformations are 3×3 matrices managed by instances of the `Transform` object. Transformations are applied by using the appropriate methods to associate them with windows, frames, etc.

Part editors do not usually need to implement their own `Transform` objects, but that option is available so that nonlinear transformations that cannot be represented by a 3×3 matrix can be created.

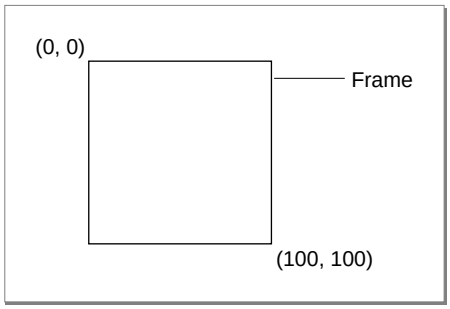
4.2.15 Coordinate Spaces

This section discusses the different coordinate spaces for the different OpenDoc objects. The examples in this section assume a coordinate system with the origin in the upper left, x increasing to the right and y increasing downward.

4.2.15.1 Frame Coordinate Space

The **frame coordinate space** is the coordinate system defined by the specification of the frame shape. The frame shape is the basis for the layout and drawing of embedded parts. Figure 26 shows a frame shape is defined as a rectangle with coordinates of (0, 0) and (100, 100).

Figure 26 Frame shape (in its own frame coordinates)



In the coordinate space of this frame in this graphics system, an item at its upper left corner is at (0, 0). Most shapes passed back and forth between embedded parts and their containing parts are expressed in terms of the frame coordinate space.

Note that the shapes describing the facet associated with a frame are described in the same coordinate space as the frame; that is, they are in frame coordinates. Thus, in this example, if the facet corresponded exactly to the frame (which is common), it would have coordinates of (0, 0) and (100, 100).

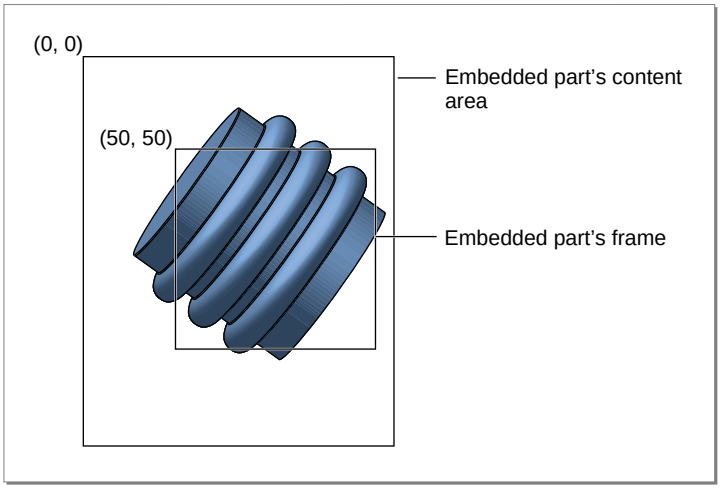
4.2.15.2 Content Coordinate Space

The **content coordinate space** of a part is the coordinate system that defines locations in the part's content area. It is the local coordinate space of the part itself.

The internal transform of a part's display frame is used to define the scrolled position of the part's contents within the frame. Applying the display frame's internal transform to a point in content coordinate space converts it to frame coordinate space. Conversely, applying the inverse of the internal transform to a point in frame coordinate space converts it to content coordinate space.

For example, suppose that Figure 27 shows the entire contents of an embedded part, a portion of which is to be displayed in a frame. If the embedded part's display frame has the dimensions shown previously (in Figure 26), and if the frame's internal transform specifies an offset value of (–50, –50), the frame would appear in relation to its part's content as shown in Figure 27. Only the portion of the part within the area of frame shape is displayed when the part is drawn.

Figure 27 Frame shape (in content coordinates of its part)



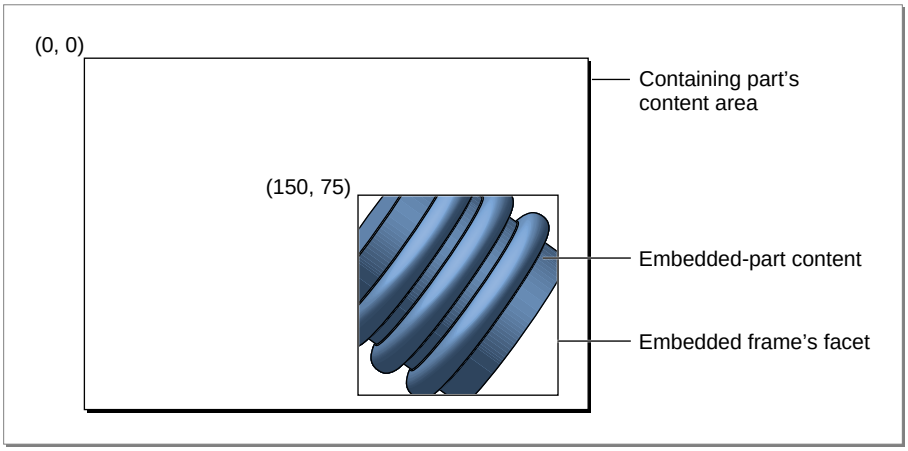
Application of the internal transform in this case means that a point at (50, 50) in content coordinates—the upper left corner of the display frame—is at (0, 0) in frame coordinates. Conversely, a point at (–50, –50) in frame coordinates is at (0, 0) in content coordinates.

4.2.15.3 Converting to the Coordinates of a Containing Part

Just as the internal transform of a frame defines the scrolled position of the content it displays, the external transform of that frame's facet defines the position of the frame within its containing part.

Applying the external transform of a frame's facet to a point in frame coordinate space converts it to the content coordinate space of the containing part. For example, suppose the facet and frame of the embedded part in Figure 27 have the same shape, and suppose further that the facet's external transform specifies an offset of (150, 75). In relation to the containing part's content area, the embedded part would appear as shown in Figure 28.

Figure 28 Frame shape (in content coordinates of its containing part)



Application of the external transform in this case means that a point at (0, 0) in embedded-frame coordinates—the upper left corner of the embedded part's frame—is at (150, 75) in containing-part content coordinates.

(Conversely, applying the inverse of the embedded facet's external transform to a point in content coordinate space converts it to the embedded frame's coordinate space. Thus, in Figure 28, a point at (–150, –75) in embedded-frame coordinates is at (0, 0) in the content coordinates of the containing part).

Transformations other than offsets are applied in the same manner; the embedded part and its frame can be scaled, rotated, or otherwise transformed within the containing part by application of the facet's external transform.

To convert from the content coordinates of an embedded part to the content coordinates of its containing part therefore requires the application of two transforms: the internal transform of the embedded part's display frame, followed by the external transform of that frame's facet. For the example shown in this section, one can see by inspection that the point (50, 50) in the content coordinates of the embedded part (the origin of its display frame) converts to the point (150, 75) in the content coordinates of the containing part. One could also calculate that the point (0, 0) in the contents of the embedded part (its upper left corner) converts, by application of both transforms, to the point (100, 25) in the contents of the containing part (outside of the embedded frame and therefore not drawn).

In general an embedded part is not often concerned with its location within the content area of its containing part. It is, however, most often interested in its content's location—and its frame's location—on the canvas or in the window in which it is drawn, as described next.

4.2.15.4 Canvas Coordinates and Window Coordinates

When a part draws its contents, or when it draws adornments to its frame such as scroll bars, it is the part's responsibility to properly position what it draws on its canvas. In setting up the canvas's platform-specific drawing structure, information that tells it where the drawing will take place in terms of its own coordinate space is provided. OpenDoc doesn't do that for you automatically. It does, however, provide methods that make it fairly simple.

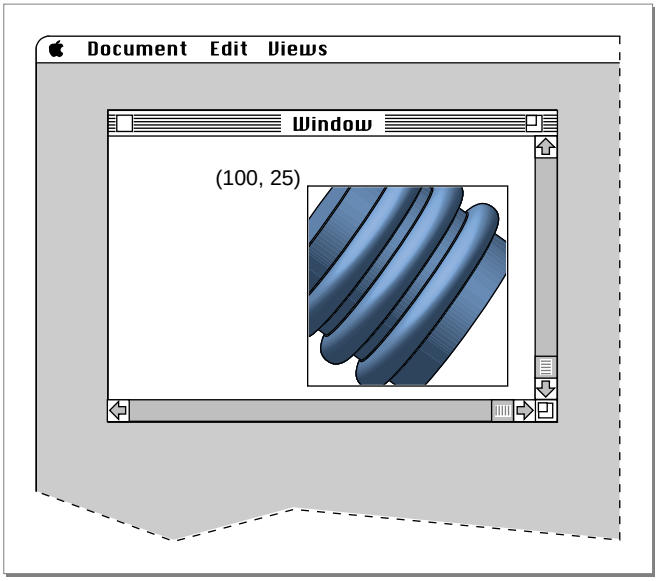
Starting from the part's content coordinate space and traverse the embedding hierarchy upward, applying in turn the part's internal transform and external transform, followed by each internal transform and external transform of all

containing frames and facets, up to a facet with an attached canvas, the part arrives at the **canvas coordinate space** or the **window coordinate space** in which the part is drawn:

- If there are no offscreen canvases in the facet hierarchy, the canvas coordinate space is the same as the window coordinate space, and calculating it involves applying all transforms up through the external transform of the root facet. This coordinate space corresponds to the device coordinates of the window canvas.
- If there are one or more offscreen canvases in the facet hierarchy, the canvas coordinate space corresponds to the frame coordinates of the first part above the part in the hierarchy whose facet has an attached canvas; calculating it involves applying all transforms up through the internal transform of the frame whose facet has the canvas. The window coordinate space in this case is unaffected by the presence of an offscreen canvas: it is still the device coordinates of the window canvas.

Figure 29 shows a simple example of converting to canvas coordinates and window coordinates. The containing part and the embedded part are the same ones as shown in Figure 28 on page 41, and the containing part is in this case the root part of the window. The internal transform of the root frame (the containing part's display frame) specifies an offset of (–50, –50), and the external transform of the root facet is identity.

Figure 29 Frame shape (in canvas coordinates and window coordinates)



Converting content coordinates to window coordinates means concatenating all the transforms up through the root frame's internal transform. If there is no offscreen canvas in Figure 29, the point (50, 50) in content coordinates (the origin of the embedded part's frame, as shown in Figure 27 on page 40) becomes the point (100, 25) in window or canvas coordinates. If there were an offscreen canvas attached to the embedded facet in Figure 29, the canvas coordinates for the origin of the embedded part's frame would be (0, 0).

Normally, all drawing is done in canvas coordinate space. That way, positioning will be correct whether or not drawing is to an offscreen canvas (which the part might not control or even be aware of). However, when drawing directly to the window to provide specific user feedback, the part must work in window coordinates.

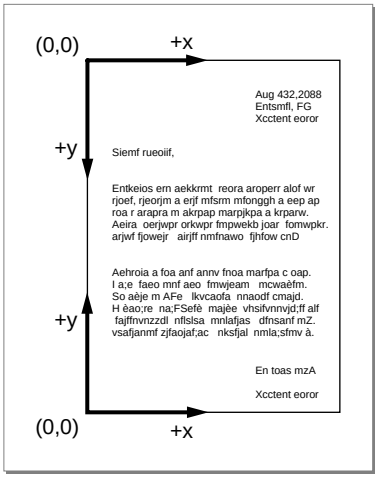
By concatenating the appropriate internal and external transforms, OpenDoc calculates four different composite transforms that can be used for positioning before drawing:

- The composite transform from the part's content coordinates to canvas coordinates is called the **content transform,** and it is applied when drawing the part's contents on any canvas. It is obtained by calling the facet's `acquire_content_transform` method.
- The composite transform from frame coordinates to canvas coordinates is called the **frame transform,** and it is applied when drawing any frame adornment on the canvas. It is obtained by calling the facet's `acquire_frame_transform` method.
- The composite transform from content coordinates to window coordinates is called the **window-content transform,** and it is applied when drawing the part's contents directly to the window. It is obtained by calling the facet's `acquire_window_content_transform` method.
- The composite transform from frame coordinates to window coordinates is called the **window-frame transform,** and it is applied when drawing any frame adornment directly to the window.It is obtained by calling the facet's `acquire_window_frame_transform` method.

4.2.15.5 Coordinate Bias and Platform-Normal Coordinates

On each platform, OpenDoc uses the platform's native coordinate system, called **platform-normal coordinates,** for stored information and for internal calculations. For example, on Macintosh and Windows platforms, coordinates are measured with the origin at the upper left (of the screen, of a shape, or of a page, for example), with increasing values to the right and downward. Some other platforms use coordinates with an origin at the lower left, with values increasing to the right and upward. Figure 30 shows how these two coordinate systems would apply to measurements on a text part's page.

Figure 30 Two coordinate systems for measuring position in a part's content



OpenDoc functions consistently on any platform, regardless of the platform's coordinate system, without need for coordinate conversion. However, a part editor that assumes a particular coordinate system will not function correctly with OpenDoc on a platform with a different coordinate system, unless it first accounts for the **coordinate bias,** or difference between its coordinate system and platform-normal coordinates.

4.2.15.5.1 Bias Transforms

Coordinate bias usually takes the form of an offset in the origin, a change in the polarity of one or more axes, and possibly a change in scale. A transformation matrix, called a **bias transform,** is applied to measurements in a part's coordinate system to change them into platform-normal values.

A part can function properly on a platform whose coordinate system is different from the part editor's assumed coordinate system if an offscreen canvas on your part's facets is installed and the proper bias transform is assigned to it. The part draw to that canvas using its own coordinates, and then transfers the (automatically transformed) results to the onscreen, platform-normal, parent canvas.

4.2.15.5.2 Content extent

In converting locations in a part's content between coordinate systems whose origins are offset, one can see from the example in Figure 30 that the vertical extent of the content (in essence, the height of the part's page) represents the offset between the two coordinate origins. This **content extent** is the offset needed in the bias transform that performs the conversion.

Because a part may at any time be drawn on a canvas that has a bias transform attached, it should at all times make the value of the content extent available. At initialization and whenever the part's content extent changes (such as when a new page is added), the display frame's `change_content_extent` method should be called so that the frame always stores the proper value. A caller constructing a bias transform can obtain the current content extent of a part by calling the `get_content_extent` method of the part's frame.

4.2.15.5.3 The bias_canvas Parameter

The classes `Frame` and `Facet` include several methods, such as `change_internal_transform` and `contains_point`, that specify shapes or calculate positions on a canvas. Because these calculations necessarily assume a coordinate system, the methods include a parameter, `bias_canvas`, that allows a canvas to be specified whose attached bias transform is to be used to convert between the part editor's coordinates and platform-normal coordinates. Thus, once an offscreen canvasses set up for drawing in the part's own coordinate system, it can also be used to make sure that all point, frame, and facet geometry are properly converted for the part.

4.2.16 Shapes

OpenDoc defines areas using shapes. Four shapes are commonly used: the **frame shape** is the shape of a frame, the **clip shape** is the unobscured area of a facet, the **used shape** is the area of a part's frame that has content to display, and the **active shape** is the area of a facet within which a part responds to mouse events.

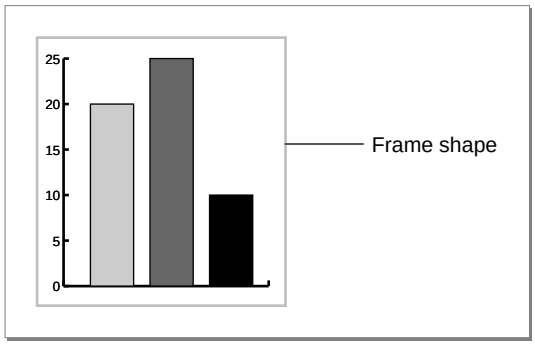
Shapes are managed by instances of the `Shape` object. Shapes can be either rectangular, polygonal, or non-geometric. Non-geometric shapes are not allowed for some uses of `Shape` objects. `Shape` object methods can be used to construct shapes using shape algebra.

4.2.16.1 Frame Shape

The frame shape represents the fundamental contract between embedded part and containing part for display space. The other drawing-related shapes are variations on the frame shape, and all are defined in the same coordinate system as the frame shape.

A frame shape is commonly rectangular, as shown in Figure 31, but it can have any kind of outline. An embedded part can request a nonrectangular frame shape; for example, a part that displays a clock face might request a round frame shape. It is the containing part's right to comply with or deny that request, and it is also the containing part's responsibility to draw that frame's selected appearance, including resize handles if appropriate.

Figure 31 A frame shape



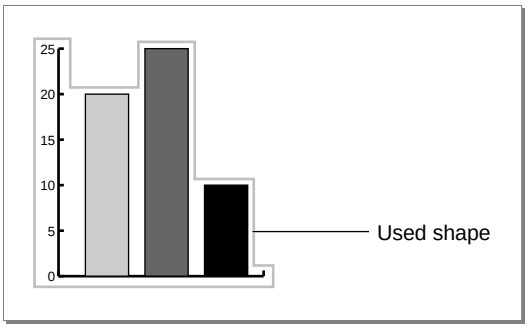
The frame shape is defined by the containing part and is stored in the frame object. A containing part can set an embedded frame's frame shape by calling the frame's `change_frame_shape` method. Any caller may access the frame shape by calling the frame's `get_frame_shape` method.

By convention, a containing part should use the frame shape when drawing the selected frame border of an embedded part.

4.2.16.1.1 Used Shape

An embedded part that needs a nonrectangular shape to draw in, or whose needed frame shape changes often, or that wants to allow the containing part to be able to draw within portions of its frame, need not negotiate for an unusual frame shape or continually renegotiate its frame size. It can instead define a **used shape,** a shape with which it tells its containing part which portions of its frame shape it is currently using. In Figure 32, for example, the embedded part retains a rectangular frame shape, but defines a used shape that covers only the content elements it draws. The containing part can then use that information to, perhaps, wrap its content to the used portions of the embedded part.

Figure 32 A used shape



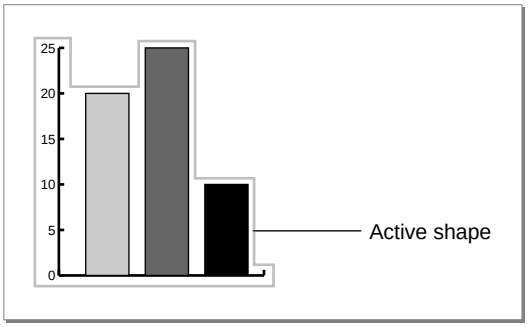
The used shape is defined by the embedded part, it is specified in frame coordinates, and it is stored in the frame object. An embedded part can set its display frame's used shape by calling the frame's `change_used_shape` method. Any caller may access the used shape by calling the frame's `acquire_used_shape` method.

Note that it does not make sense for the used shape to extend beyond the edges of the frame shape, since the clip shape is based on the frame shape and no drawing occurs outside of the clip shape.

4.2.16.1.2 Active Shape

A facet's **active shape** is the area within a frame in which the embedded part is willing to receive geometry-based (mouse) events. The active shape of a facet is often identical to either the frame shape or the used shape of its frame, as shown in Figure 33. The active area of a part is likely to coincide with the area it draws into; events within the frame shape but outside of the used shape are better sent to the containing part, which may have drawn in that area.

Figure 33 An active shape



The active shape is defined by the embedded part, it is specified in frame coordinates, and it is stored in the facet object. An embedded part can set the active shape of its display frame's facet by calling the facet's `change_active_shape` method. Any caller may access the active shape by calling the facet's `acquire_active_shape` method.

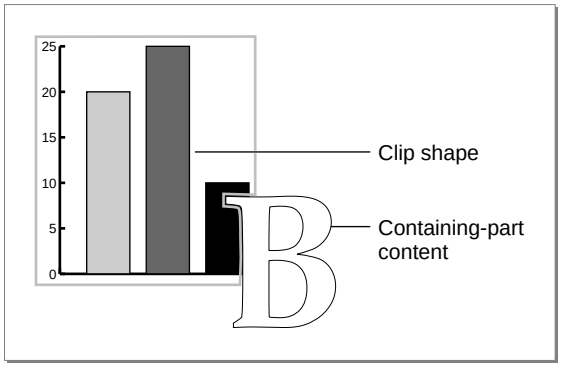
Note that the effective active shape—the active shape as perceived by the user—is the intersection of the active shape and the clip shape (described next); events within the area of obscuring content or other frames that block the active shape are not passed to the embedded part.

OpenDoc uses the active shape when drawing the active frame border of an embedded part.

4.2.16.1.3 Clip Shape

A facet's **clip shape** defines the portion of a frame that is actually to be drawn. The clip shape of a facet is commonly identical to the frame shape of its frame, except that it may be modified to account for obscuring content elements or overlapping frames in the containing part, as shown in Figure 34.

Figure 34 A clip shape



The clip shape is defined by the containing part, it is specified in frame coordinates, and it is stored in the facet object. A containing part can set the clip shape of an embedded frame's facet by calling the facet's `change_clip_shape` method. Any caller may access the clip shape by calling the facet's `acquire_clip_shape` method.

4.2.17 Data Transfer

Transfer of data between parts is fundamental to OpenDoc. Data transfer facilities include a clipboard, dragging and dropping, and linking. These facilities have several concepts in common.

4.2.17.1 Incorporation vs. Embedding

A part receiving external data can treat it in one of two ways. It can either embed the data as a separate part, or it can incorporate it into its intrinsic content. OpenDoc and the user make that decision, based on the part's own native part kinds and part categories, and the part kinds and part category present in the data.

4.2.17.2 Data Translation

Translation may be required to transfer data to a part. Translation during data transfer is analogous to the translation that can occur when a document is first opened, and depends on the availability of translation facilities that can convert data of one part kind to another. OpenDoc presents translation possibilities to the user, who then decides whether to translate and what translator to apply.

4.2.17.3 Promises

Data transfer can make use of promises. A **promise** is a specification of data to be transferred at a future time. If a data transfer involves a very large amount of data, the source part can choose to put out a promise instead of actually writing the data to a storage unit. When the data is retrieved by another part, the source part must then fulfill the promise it put out. The destination part does not even know that the fulfillment of a promise is taking place; it simply accepts the data as usual.

4.2.17.4 Linking

OpenDoc supports linking, which allows a part (the destination) to include data from another part (the source). The destination data is updated when the source data is modified. The user can choose between automatic updating and manual updating that requires user intervention.

Links incorporate **change ID**s. These IDs change every time the source data is changed. These IDs can also be used to detect and prevent circular links.

Parts must use a standard format when writing their content. Source parts write a reference to the storage unit of the `LinkSource` object. Destination parts write a reference to the storage unit of the `Link` object along with a flattened `LinkInfo` structure. This structure consists of a fixed-length, byte-for-byte copy of the `LinkInfo` structure, fol-

lowed by a variable-length part-kind field. The part-kind field consists of a four-byte length followed by an ISO string that defines the part kind of the link's content.

4.2.18 Clipboard Transfer

The Clipboard is a mechanism for transferring data within and across documents. The Clipboard commands Cut, Copy, and Paste operate on embedded parts as well as intrinsic content. The data copied to or pasted from the Clipboard can contain any number of parts, of any part kinds. The parts may be displayed in frames or represented as icons.

- The Paste and Paste As commands can either embed parts or incorporate data as intrinsic content.
- The Paste and Paste As commands can create a new part when it embeds intrinsic content of one part kind into a part of a different part kind.
- The Paste As command can create a persistent link to the source of the data being pasted.
- The Paste As command also allows the user to override decisions on incorporating vs. embedding that would normally be made by OpenDoc.

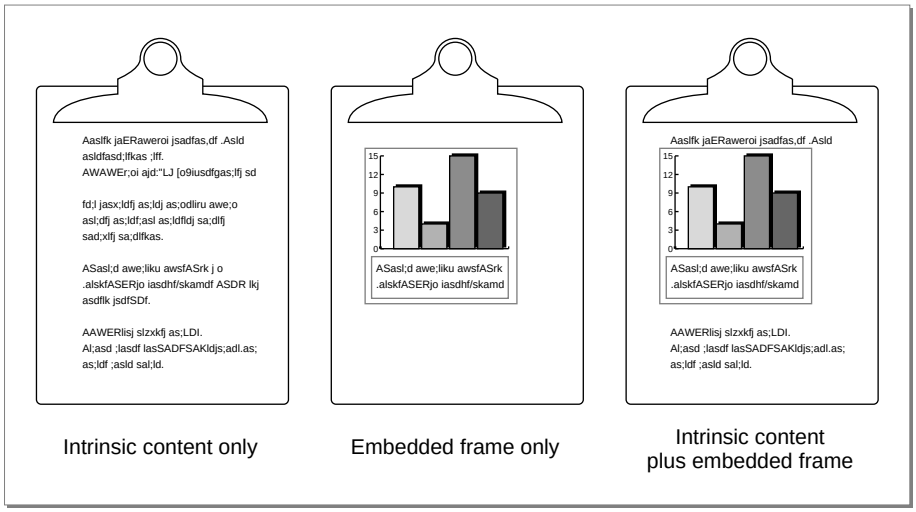
As with data stored in a document, and as with data placed in a conventional Clipboard, a part editor can store multiple formats of its data, including standard formats, on the Clipboard. That way, the user can directly incorporate (rather than embed) Clipboard data into more kinds of parts.

If the user cuts or copies a part to the Clipboard, the moved data represents the current state of the part, including any edits that have not yet been saved.

4.2.18.1 Moving Data to the Clipboard

When a part places data on the clipboard, it can place two types of data: intrinsic content and embedded parts. As shown in Figure 35, the data can be in different configurations. The Clipboard can contain intrinsic content only, or the contents of a single embedded frame (which may itself have other frames embedded within it), or a combination of intrinsic content plus embedded frames.

Figure 35 Configurations of data on the Clipboard



(Note that, if a single embedded frame is placed on the Clipboard, as in the middle illustration of Figure 35, the frame object itself is not actually written to the clipboard, although a specification of its shape is.

The storage unit for the Clipboard's intrinsic content (or its outermost part if it consists of a single frame), is called the Clipboard's **content storage unit.** It is the main storage unit for the clipboard, equivalent to a part's main storage unit. Any embedded parts on the Clipboard, and any other OpenDoc objects, have their own storage units.

Also, when a part copies data to the Clipboard, the part must place, in addition to the data itself, the capability for constructing a link back to the source of the data.

4.2.18.1.1 Copying Intrinsic Content

This is the simplest Clipboard action. A part places some of its intrinsic content—containing no embedded parts—onto the Clipboard. In summary, take these steps:

1. Acquire the `Clipboard` object through the `Session` object's `get_clipboard` method. Acquire the clipboard focus, using the `Arbitrator`'s `request_focus` method. Remove any existing data on the clipboard with the Clipboard's `clear` method, and get access to its content storage unit, using the Clipboard's `get_content_storage_unit` method.
2. Add a property (of type `prop_contents`) to the Clipboard storage unit that holds the data to place on the Clipboard. Write the data into that property, in a value specified by the data format of the intrinsic content (the part kind).
   In that same property, write additional public formats of the data as separate values, so that users can incorporate a version of the data into several different part kinds.
   Note that, instead of writing actual data, a part can write a promise to the clipboard.
3. If possible, add another property (of type `prop_frame_shape`) to the Clipboard storage unit that specifies a default frame shape, in case the data must be embedded as a separate part in its destination. In the absence of this frame shape, the destination part uses its own default frame shape for embedded parts.
4. When copying rather than cutting, create a link specification (using your draft's `create_link_spec` method), in case the user chooses to link to the data instead of pasting it in the destination. The specification contains whatever information the part chooses that identifies the source of the link; it is returned if your part's `create_link` method is called. Write the link specification onto the Clipboard as a property of type `prop_link_spec`.
5. Using the Clipboard's `get_change_id` method, get and save the Clipboard's current **change ID,** a number used to identify this particular instance of Clipboard contents. If the link specification becomes invalid before the clipboard changes again (that is, if the content at the source of the link changes), remove the link specification from the Clipboard. Examine the Clipboard's current change ID at any time and compare it with your saved change ID; if they are identical, the Clipboard has not changed.
6. Relinquish the Clipboard focus. Alternatively, wait until asked to relinquish the focus by the next requestor of the Clipboard focus.

4.2.18.1.2 Copying a Single Embedded Part

If the selection to be placed on the Clipboard consists of a single frame of an embedded part—with no surrounding intrinsic content—the data on the Clipboard should be a duplicate of the embedded part. The nature of the containing part usually has nothing to do with the operation. However, a part can write some data associated with the frame itself, if desired; see step 4.

In summary, take these steps:

1. From the embedded frame, get access to the embedded part (`acquiret_part` method). As in the previous example, focus on the clipboard, remove any existing data on it, and get access to its content storage unit.
2. Copy the embedded part to the Clipboard by cloning it.
   Call the draft's `begin_clone` method, followed by the embedded part's `clone_into` method, followed by the draft's `end_clone` method. Cloning is a three-stage process, to ensure that all objects to be cloned are identified before the cloning process actually occurs.
   Specify a copy operation (type `clone_copy`) for this clone. The cloning process copies the embedded part's data onto the Clipboard, plus any parts and frames embedded within it; cloning does not copy the part's display frame (or frames), to which the part has a weak persistent reference.
3. Explicitly write the frame shape of the embedded frame to the Clipboard. Also, to be sure that the part on the Clipboard is embedded rather than incorporated when it is pasted, add a property (of type `prop_content_is_subframe`) with a value of `TRUE` to the Clipboard.
4. Write any intrinsic data that the part wants associated with the frame (such as a drop shadow or other visual adornment). Add a property (of type `prop_proxy_contents`) to the Clipboard, and write the data into it as a value that the part recognizes. If the Clipboard part is subsequently pasted into a part that also recognizes that value, the added frame characteristics can be duplicated.
5. As in the previous example, write a link specification onto the Clipboard so that users can link to the data instead of pasting it in the destination.

6. Save the Clipboard's current change ID and relinquish the Clipboard focus, as in the previous example.

4.2.18.1.3 Copying Content That Includes an Embedded Frame

If the selection to be placed on the Clipboard consists of a combination of a part's intrinsic content plus embedded frames, follow a process that is a combination of the previous two examples: clone embedded parts and write the intrinsic content as well. These are the basic steps (for simplicity, these steps assume that a single frame is to be copied to the Clipboard):

1. As in the previous examples, acquire the Clipboard focus, remove any existing data on it, and get access to its content storage unit.
2. Copy the embedded frame to the Clipboard by cloning it. Call the draft's `begin_clone` method (specifying a copy operation), and then call the embedded frame's `clone_to` method, followed by the draft's `end_clone` method.
3. Write the intrinsic data into a property of type `prop_contents` on the clipboard. Write additional public formats of the data as separate values. (The intrinsic data will reference the frame cloned in the previous step.)
4. As in the previous examples, write a frame shape to the Clipboard, create a link specification, save the Clipboard's current change ID, and relinquish the Clipboard focus.

4.2.18.1.4 Copying Linked Content

A part can allow the user to place material on the Clipboard that consists of link destinations, link sources, or both.

To make the copy, clone each associated `Link` or `LinkSource` object to the Clipboard. Then copy the intrinsic content, which includes references to the cloned objects. These references are stored in the Clipboard's `prop_contents` property, not in a separate property. Each value in the contents property must be complete, and must not depend on other properties or other values of the contents property.

In summary, take these steps:

1. As in the previous examples, acquire the Clipboard focus, remove any existing data on it, and get access to its content storage unit.
2. As when copying a single part, clone the link objects to the Clipboard using a copy operation. Add a property (of type `prop_contents`) to the Clipboard storage unit, to hold the intrinsic content plus link references:
3. For each link source in the selection, clone the `LinkSource` object to the clipboard and assign that storage unit as the link's new source, using the link source's `clone_to` method. Write a strong persistent reference to the link source's storage unit into the intrinsic content placed on the Clipboard.
4. For each link destination in the selection, clone the `Link` object to the clipboard, using the link's `clone_to` method. Write a strong persistent reference to the link's storage unit into the intrinsic content placed on the Clipboard. Also write the link information (type `LinkInfo`) for the link into the intrinsic content placed on the Clipboard.
5. Write the remainder of the part's data to the clipboard, using the data format of the intrinsic content (corresponding to the part kind). Also write additional public formats of the data as separate values. (Those versions need not necessarily include links.)
6. As in the previous examples, write a frame shape to the Clipboard, create a link specification, save the Clipboard's current change ID, and relinquish the Clipboard focus.

4.2.18.1.5 Removing a Link Specification From the Clipboard

If a part copies some of its content to the Clipboard, and the user then modifies the original content in the part without first having performed a paste, the Clipboard data no longer matches the source data in the part. The potential link represented by the link specification written to the Clipboard is therefore no longer valid.

To prevent the user from creating an invalid link, the source part must remove the link specification in that situation. By saving the change ID of the Clipboard whenever data is copied to it, that ID can be checked against the current change ID of the Clipboard whenever the source data changes. If the IDs match, the Clipboard contains the same data that was placed in it, and remove the link specification should be removed.

Follow these steps to remove a link specification:

1. Acquire the Clipboard focus and get access to the Clipboard's content storage unit.
2. Focus the storage unit on the link-specification property (type `link_spec`), and remove that property (using the storage unit's `remove` method).

3. Relinquish the Clipboard focus.

4.2.18.2 Pasting Data from the Clipboard

When a Paste command places data into a document, how that data is handled depends on how the part kinds within the clipboard data relate to the part kind of the destination part (the part at the location of the paste).

4.2.18.2.1 Default Conventions

In the absence of other instructions, OpenDoc follows specific conventions when pasting data from the Clipboard or when dropping data during a drag-and-drop operation. Because pasting and dropping are such similar actions, this section discusses conventions for both.

- If the transferred data (Clipboard data or dragged data) is entirely of the same part kind as the destination part, the data is simply incorporated into the content of the destination part. Clipboard data is pasted at the insertion point; dragged data is dropped at the point where the user clicks.
- If the highest-fidelity representation of the transferred data is of a different part kind from the destination part, but the destination part editor can directly read another representation that is in the data, the destination part editor reads the data and incorporates it into the intrinsic content of the destination. Usually, this means that both part kinds must be of the same part category.
- If the transferred data is of a different part kind from the destination part, and if the destination part editor cannot read any of its representations, the destination part editor creates a new part of the same part kind as the transferred data, inserts the transferred data into the new part, and embeds the new part in the destination.
  If a part refuses to accept a paste because the data contains a part of a kind not permitted by the part's inclusions list, OpenDoc displays a dialog box, notifying the user of the restriction.
- If the transferred data represents a single embedded frame that was cut or copied (or dragged), the destination part editor embeds the data as a separate part into the destination part, regardless of whether its part kind is different from or identical to that of the destination.
  Note that a pasted or dropped part takes on the view type that its containing part prefers it to have.
- If the transferred data represents intrinsic content plus one or more embedded parts that was cut or copied (or dragged), the destination part can do one of two things:
  - If the intrinsic data on the Clipboard (or in the drag-and-drop object) is not of the same part kind as the destination, the destination part editor creates a new part of the same part kind as the intrinsic data, inserts the transferred data into it, and embeds the new part (and its embedded parts) in the destination.
  - If the intrinsic data on the Clipboard is of the same part kind as the destination, the destination part just embeds the Clipboard data into the destination's intrinsic content.
- If the destination part has an icon view type, it can still accept a drop (but not a paste) from the user, if it makes sense for the part. Place the dropped item inside the part at some location. If more than one item is dropped, the destination part should preserve the spatial and other relationships among the dropped items, if possible.

4.2.18.2.2 Pasting As…

As noted in the previous section, pasting may be accomplished by embedding, by incorporating, or by translating and then incorporating or embedding the pasted data. The Paste As command in the Edit menu allows the user to override OpenDoc and specify which action is to occur. It also allows the user to create a link to the source of the Clipboard data.

The user can select the following options:

- Paste with link. A link is created between the source and destination data. Note that this option is not available to the user unless both the source document and the destination document are open. A part responds to this option as described in the section "Incorporating Data and Creating a Link" on page 51.
- Update (a link) Automatically/Manually. Link updates can be either automatic or manual; this choice sets which method is to be used.
- Kind (of pasted data). OpenDoc chooses the default part kind of the data to be pasted into the destination. The user can use this option to override that choice, explicitly specifying a destination part kind and, if necessary, a

translator to perform the conversion. Only one translation step is allowed, meaning that conversion must be directly from the part kind on the Clipboard to the part kind selected by the user. Note also that translation applies only to the outermost part of Clipboard data; parts embedded within the Clipboard data are not translated. The part kinds presented to the user in this option can vary, depending on whether the user has selected either of the following two options:

  - Incorporate regardless. If the user chooses this option, the destination part editor is expected to translate the source data into the destination part kind, even if it means great loss of information (such as converting text to a bitmap). This option is not available if the destination cannot incorporate the data, even after translation.
  - Embed regardless. If the user chooses this option, the source data must be embedded in the destination as a separate part, with a view type chosen by the user, even if incorporation is possible. If the destination part's restrictions list forbids embedding data of the part kind of the Clipboard data, this option is not available.

- Based on the results of the Paste As dialog, a part will either embed or incorporate the Clipboard data, either accept it as it is or translate it, and either create a link to its source or not. The following sections outline the steps performed to take those actions.

4.2.18.2.3 General Pasting Procedure

Most paste operations should clone the content storage unit of the Clipboard into a new storage unit in the destination part's draft. The destination part generally lets OpenDoc transfer links and embedded parts into the destination. Once the transferred data has been placed in the draft, the destination part can embed or incorporate, as appropriate.

4.2.18.2.4 Incorporating All Data as Intrinsic Content

If the clipboard contains only data directly readable by a part, and there are no embedded frames and no links or link sources, *and* if the part editor does not support embedding or linking, OpenDoc does not define how to read the data and insert it into your part. This is the very simplest possible pasting procedure. Here are the steps:

1. As when copying to the Clipboard, acquire the Clipboard focus and get access to its content storage unit.
2. Copy the data from the Clipboard's content storage unit and adding it to the part data.
3. Notify all containing parts that the content has changed, by calling the display frames' `content_changed` method.
4. Call the draft's `set_changed_from_prev` method, to mark it as dirty so that the part will be able to write its revised content out to persistent storage when the draft is closed.
5. Relinquish the Clipboard focus.

4.2.18.2.5 Incorporating Data and Creating a Link

A part can incorporate data from the clipboard and at the same time create a link to its source.

The part needs to associate a link-info structure (type `LinkInfo`) with each link destination it maintains. The link-info structure holds information, such as size, creation date, and change ID, about the content at the destination of a link. If the link destination is later moved or copied, this information should be transferred along with it, so the new destination can maintain the same characteristics as the original.

In addition to setting up the clipboard and reading in the intrinsic data as shown in the previous section:

1. Focus the Clipboard storage unit on the link specification in the Clipboard. Use the draft's `create_link_spec` method to create and initialize the link specification, then call the link specification's `read_link_spec` method to have it read itself from the Clipboard.
2. Pass the link specification to the draft's `get_link` method to construct the link object from the link specification.
3. Initialize a link-info structure with the change ID, creation and modification dates, part kind, and auto-update setting of the link.
4. Embed or incorporate the link into the part. Read its data, translate it if necessary, and display it according to its view type (if embedding).
5. If auto-updating is specified, call the link's `register_dependent` method.
6. Clean up as shown in the previous section. Call the display frames' `content_changed` method, call the draft's `set_changed_from_prev` method, and relinquish the Clipboard focus.

Note that the destination part must be able to draw a border around the link content area when asked to do so.

4.2.18.2.6 Embedding Data as a Single Part

Embedding data from the Clipboard as a single part mirrors and continues the process for putting a single embedded part onto the Clipboard, as described under “Copying a Single Embedded Part” on page 48. Basically, clone the part into the draft, create an embedded frame (or frames) for it, and create facets for those frames. In summary, take the following steps:

1. Acquire the Clipboard focus and get access to the Clipboard’s content storage unit.
2. Copy the embedded part from the Clipboard into the draft by cloning it. Specify a paste operation (type `clone_paste`) for this clone. Call the Clipboard draft’s `begin_clone` method, followed by the Clipboard’s content storage unit’s `clone_into` method, followed by the Clipboard draft’s `end_clone` method.
3. If the Clipboard’s content storage unit contains a property of type `prop_proxy_content`, that property contains any **proxy content** that the part’s original containing part wanted associated with the frame, such as a drop shadow or other visual adornment. (Note that this property is absent if the Clipboard content includes intrinsic content as well as an embedded frame.) Focus the Clipboard content storage unit on the `prop_proxy_content` property and read in the information. Note that the part must understand the format of the proxy content in order to use it, and it becomes part of the part’s intrinsic content to be associated with the frame.
4. Create a shape object for the embedded part’s frame. Obtain the frame shape from the Clipboard, if it exists, and read that information into the frame shape; otherwise, assign a default frame shape.
5. For each of the part’s display frames that will contain the embedded part, create an embedded frame and assign the embedded part to it.
6. Relinquish the Clipboard focus.

4.2.18.2.7 Embedding Data as a Single Part and Creating a Link

A part can also embed linked content in a destination part, rather than incorporating it. A part creates an embedded part with content from the link, exactly as if content were embedded without a link. The only difference is that, every time the link is updated, the destination part discards the embedded part and creates a new embedded part from the new link content.

The embedded part is not involved in the maintenance of the link; the link border appears around the embedded part's frame and is drawn by the destination part.

4.2.18.2.8 Incorporating Content That Includes an Embedded Frame

If the user pastes intrinsic data that can be incorporated into a part, but that data also includes one or more embedded parts, perform a process like the sum of the previous two examples. The steps to take are similar to the following (for simplicity, these steps assume that a single frame is embedded in the Clipboard data, and that the content is not linked):

1. Acquire the Clipboard focus and get access to the Clipboard’s content storage unit.
2. Copy the intrinsic content from the Clipboard into the draft by cloning it. Even when content ends up being incorporated into the part, cloning it into the destination draft ensures that, if it is linked content, its semantics are preserved. For example, if the content is a link source, the cloning ensures that the paste doesn’t result in the creation of two sources for a single link.
   The clone operation also copies the embedded part content.
3. Incorporate the cloned content (other than the embedded part). How a part handles this is not defined by this standard. The incorporating routine might replace the current selection, if any, with the intrinsic content, and define a frame shape and location for the newly embedded part. Use the frame shape from the Clipboard, if it exists.
4. Create an embedded frame with the frame shape and assign the embedded part to it.
5. Relinquish the Clipboard focus.
6. Release the storage unit for the cloned data that was incorporated.

4.2.18.2.9 Assimilating Links

If the user pastes information into a part that consists of links to or from data that can be incorporated into the part, it can retrieve that data, incorporate it permanently into the part, and redirect all link source objects to refer to it.

Here are the steps to take, after acquiring the Clipboard focus and cloning the content storage unit into a draft:

1. If the new storage unit contains intrinsic data of a part kind readable by the part editor, focus the new storage unit

on that data. (If it does not, but it does contain a degraded format—such as plain text—that can be read but does not support links, the part can still extract that data and incorporate it directly, without maintaining the links.)

2. For each link source in the Clipboard data, read the content-specific information (assumed to be understandable by the part editor) that associates the link-source object with its content. Call the link's `set_source_part` method to associate it with the part, and add the link source to whatever private structures the part maintains for link sources.
3. For each link destination in the Clipboard data, read the link-info data and store it in the part, read the content-specific data (assumed to be understandable by the part editor) that associates the link object with its content, and add the link to whatever private structures that the part maintain for link destinations.

The part can then retrieve and incorporate any remaining intrinsic content in the storage unit, and create embedded frames if necessary as shown in the previous section. As usual, clean up by calling your display frames' `content_changed` method, calling your draft's `set_changed_from_prev` method, relinquishing the Clipboard focus, and releasing the storage unit for cloned data that you incorporated.

4.3 The Document Shell

OpenDoc uses a runtime model significantly different from the classic personal computer model. Commonly on personal-computer platforms, each application is a monolithic process. This process provides the address space in which the application code executes, as well as the memory for data files opened by that application. The operating system spawns a process for an application and provides support services for it.

In OpenDoc, a document no longer has a single type, but is instead composed of many parts that may be of different types. Each part is manipulated by associated part editor code as opposed to using a single monolithic application to manipulate all known data types. What the user perceives as a single application is in reality a collection of application code pieces acting cooperatively.

Thus, the document content controls the collection of application code pieces that appear as a single application process to the user. The document controls the management of document-wide and draft-wide tasks such as handling events, managing files, printing, and interacting with system menus, dialog boxes, and so on. In OpenDoc, this behavior is provided by the **document shell**.

The document shell's responsibilities include the following:

- It creates and initializes the `Session` object, the storage system, and the **window state**.
- It opens a document (as instructed by the user) from storage, and provides screen real estate for it.
- It accepts user events, and passes them to the OpenDoc dispatcher.
- It uses the OpenDoc arbitrator to negotiate shared resources among parts.
- It handles most items on the Document menu.
- It manages document wide memory usage.
- It binds part editors and causes them to be executed, as needed, to allow display and manipulation of the parts of the document. To **bind** a part editor means to associate the editor code with the appropriate part data in the document.
- It creates new documents from stationery.

4.3.1 Dynamic Linking

The storage of the executable code of each part editor, and the mechanism used to collect part editors into an executable collective is system dependent. Two types of systems are common:

- Part editors are dynamically linked with the document shell into a single process, and,
- The document shell is a process that executes in concert with part editor processes.

OpenDoc requires that the platform provide a dynamic linking capability to allow the appropriate executable code to be added to the runtime environment of a document since there is no way of knowing what part editors may be needed to manipulate the contents of a document before it is opened. The dynamic linking capability must:

- allow code in one module to call code in another independent module, and
- allow code in independent modules to access global variables.

Note that there many ways that this capability can be implemented, including dynamically linked libraries and remote procedure call interfaces.

4.3.2 Binding

At runtime, part **binding** is the process of assigning a part editor to a given part. The document shell uses the `Binding` object to accomplish this task**.** The `Binding` object combines information provided by part editors, preferences specified by users, and part-kind information stored with parts, and chooses an editor for each part accordingly. If there is no suitable part editor available, the document shell binds the **editor of last resort** to the part data. This part editor does not allow editing of the part, and displays only a gray rectangle representing the area of the part's frame. It does, however, allow the user to specify a translation of the part to an editable part kind, if such a translation is possible.

The binding object uses the COSS naming service to manipulate the information it uses to make its decisions.

OpenDoc provides for the use of data translation to facilitate finding an editor for a part. If there is no editor that can read the data of a given part, and if the platform on which OpenDoc is running provides translation services, and if a translator is available that can convert from a part kind present in the part to a part kind readable by an available editor, the document shell offers the user the option of applying the translation and then binding the editor to the part. The document shell makes use of the `Translation` object during this process; the `Translation` object performs another kind of binding process to bind a translator to the part to be translated. The binding performed by the translation object is very similar to that of the binding object, except that two data formats (source and destination) are involved, instead of one.

Part binding occurs in the following situations:

- When a draft is opened, the document shell binds a part editor to each visible part of the draft.
- As new parts are added to a document, part editors are bound to them.
- Drawing a part requires part binding, if an editor has not already been bound to the part. Embedded parts within a selection may not have to be bound, as long as they are being manipulated as a whole.
- When a document is saved or copied, part binding is necessary (and will generally have occurred) for any parts whose data has changed since the draft was opened. If a part has not been changed, there is no need to bind a part editor to it.

Translator binding occurs in these situations:

- If translation is needed for a part editor to handle a part when a draft is opened.
- Whenever translation occurs (such as when the user specifies a new editor for an existing, open part).
- When pasting or dropping data into a document.

4.3.2.1 Part Kinds Stored in a Part

OpenDoc uses two kinds of information for the binding of parts:

- Information included in stored parts about the data formats in them.
- Information included in part editors about the data formats that they handle.Note that many different part editors can operated on the same part kind. For example, SurfWrite 3.0, SurfWrite 3.01 Pro, and SurfWrite 3.3 might all create parts whose part kind is "SurfWrite IntlText".

See Chapter 2.2.1.2, "Part Data Types," for more details.

A part is not limited to one part kind; it can have multiple stored representations of its data. Different representations are stored in the contents property of the part's storage unit, as values with different part kinds, arranged in order of fidelity. **Fidelity** refers to the faithfulness of a given representation to a part editor's native, or preferred, part kind. For

example, assume that a part created with your SurfWrite 3.01 Pro part editor is stored as three separate representations with the following part kinds: “SurfWrite IntlText”, “SurfWrite StyledText”, and “text”. The highest-fidelity representation is “SurfWrite IntlText”, which represents the native format of the part editor; the “SurfWrite StyledText” representation is an older, simpler format that lacks some of the latest SurfWrite features; and the “text” representation is plain, unformatted text.

4.3.2.2 Part Kinds and Categories Supported by an Editor

Part editors are not necessarily confined to manipulating a single part kind. The hypothetical SurfWrite 3.01 Pro editor, for example, may be capable of reading and writing data of SurfWrite 1.0 and plain text format, as well as its own format.

The method for defining the part kinds that an editor supports differs among platforms. An editor includes a database that lists, in fidelity order, the part kinds that the editor can read and write.

Every time a part editor writes a part to storage, it orders the stored representations in the fidelity order natural to the editor. It may need to store different representations, in a different order, from those that were present in the part when it was first read in. For example, if a SurfWrite 1.1 part editor supports only “SurfWrite StyledText” and “text” formats, and a part is opened whose highest-fidelity part kind is “SurfWrite IntlText” but which also contains a “SurfWrite StyledText” representation, the part editor needs to make sure that “SurfWrite StyledText” is the first (highest-fidelity) value in the contents property of the part’s storage unit when the part is subsequently written. A “text” representation might also be included, but a “SurfWrite IntlText” version would not (and could not) be included.

Part category is a typing scheme similar to part kind, except that it defines only a broad classification of the data manipulated by a part editor. Like part kinds, part categories are ISO strings; they have designations such as “text”, “graphics”, “time”, or “sound”. All parts created by a part editor are assigned its part category, which cannot be changed.

A part editor must specify the part categories of the part kinds it supports. Although the specific format differs among platforms, an editor includes a database that lists, for each part kind that it can read and write, the part category (or categories) that that part kind belongs to. (A part kind can correspond to more than one part category; for instance, a plain “text” part kind could be considered to be part of both “text” and “styled text” part categories, if such categories were defined.)

Editor names, part kinds, and part categories are all manipulated by OpenDoc as tokenized ISO strings, but displayed to the user as international strings that can be in any script or language. Part editors supply both ISO strings and international strings for its editor name, part kinds, and part categories.

4.3.2.3 User-Preferred Part Editors

The most obvious binding for a part is to the editor that created it. However, the highest priority binding that the document shell uses is to what the user considers the “preferred editor” for any given part kind. For every part editor installed on a user’s system, OpenDoc maintains a database that specifies the part kinds for which that editor is the preferred editor. OpenDoc sets the default preferred editor for each part kind, but the user can change the setting, in order to use, for example, a single favorite text editor for all text processing, or a single favorite graphics editor for all bitmap editing.

At runtime, when a given part is to be read in, OpenDoc looks for the preferred editor for the highest-fidelity part kind in the stored part. If it finds it, the binding object binds it to the part.

4.3.2.4 Binding to Available Part Editor

If there is no specified preferred editor for the highest-fidelity part kind of a part, that usually means that the editor that created the part is not on the user’s system and the user has not specified a preferred editor for that part kind.

In that case, the binding object examines, in order from highest fidelity to lowest, each of the representations of the stored part, seeking to match that part kind with a part kind supported by an editor on the user’s system. (If the document containing the part has an inclusions list, the binding object only examines editors that are on that list. As soon as a match is encountered, that editor is bound to the part.

The part kind used by a part editor can be changed either explicitly by the user, or implicitly if a part editor has a different preferred kind that the one last used to manipulate the part data.

4.3.2.5 Binding with Translation

If no editor is available on the user's machine that can handle any of the part kinds in a stored part, it can still be possible to find an editor, if any of the part's kinds can be translated into a part kind for which an editor does exist.

The OpenDoc translation object, implemented on each platform as a subclass of `Translation`, is a wrapper for platform-specific translation services. The document shell uses the translation object to find out what kinds of translations are available on the user's system. When a part is to be read and no editor can handle its data, the binding object examines each part kind in turn, again from highest fidelity to lowest fidelity, and determines from the translation object what new part kinds (supported by available editors) that the original part kind can be translated to. The document shell then presents those possible part kinds to the user. If the user picks one, the binding object binds the translator code to the part data and performs the translation. Then it binds the translated data to its new part editor.

4.3.2.6 Binding to Editor of Last Resort

If there is no available editor and no possible translation for a part, or if the user chooses not to substitute part editors, the part remains unviewable and uneditable, and its set of part kinds does not change. In such as instance, the document shell still binds the part to an editor, so that the document that the part belongs to can be opened. The editor used is the OpenDoc **editor of last resort,** and it may do nothing more than display an outline of the part's frame. (OpenDoc provides the name (part kind) of the part editor that is needed to display the highest-fidelity version of the part in the Part Info dialog box.)

The editor of last resort never modifies the part it displays; it does not change the part kinds of the values or their order in the part's storage unit.

4.3.3 Handling User Events

The document shell receives all user events directly and ensures that events intended for particular parts get to the proper part editor.

The document shell passes all events to the dispatcher for dispatching to the appropriate part without first classifying them. The dispatcher rejects events that it can't identify, returning them to the document shell to handle.

To dispatch an event, the dispatcher locates the appropriate dispatch module for the event, and asks it to dispatch the event. The dispatch module actually distributes the event to the appropriate part or document, in cooperation with the window state and the arbitrator, using the facet hierarchy.

The document shell also provides information to part editors about mouse tracking, and also supplies the basic menu bar used by OpenDoc documents and parts.

4.3.4 Handling the Document Menu

The document shell creates the standard document menu, and it handles most items in it. Items not handled by the document shell are handled by the root part or the active part.

Figure 36 shows a sample OpenDoc document menu. If a part editor adds items or otherwise modifies this menu when it part is active it should follow the human interface guidelines for the target platform.

Figure 36 The Document menu

The following subsections discuss the actions triggered by document menu selections.

4.3.4.1 Creating a New Document

When the user selects New, the document shell creates a new, blank document whose root part is of the same part kind as the root part of the active window and opens it in a separate document window.

4.3.4.2 Opening a Selection

When the user selects Open Selection, the document shell opens the selected part.

4.3.4.3 Opening a Document

When the user selects Open Document, OpenDoc instantiates a document shell. The document shell instantiates the session object, which in turn instantiates the globally available OpenDoc objects, such as the arbitrator and the window-state object. The document shell then opens the document file, and reads the document object and its most recent draft. The document shell then reads in the draft's window state, and reconstructs the document's windows. For each window, it reads in the root frame and constructs the root facet.

The root frame for the window reads in the root part. The root part determines which of its embedded frames are visible, reads them in and constructs facets for them, by calling the root facet's `create_embedded_facet` method. The embedded frames read their own parts (if they are currently visible), and the root part calls the `facet_added` method of each embedded part, to allow it to in turn read in its embedded frames and create facets for them, and so on until all visible frames, parts, and facets have been read from storage.

The document shell uses the window state to reconstruct the user-interface state that was stored in the document. If a frame was active when the document was last saved, and if the part editor for that frame has stored that information (perhaps in the frame's part info field), the part editor can now retrieve that information, reactivate itself, restore the selection (if it was saved also), and display any user-interface elements like palettes that are associated with the active part.

Once all of the necessary objects are read in, the document shell asks each facet to draw itself; the part editor of the facet's frame draws the part content that is visible in the facet.

4.3.4.4 Insert

The Insert selection is used to select a document and embed it as a part into another part. The command information is passed to the embedding part via its `handle_event` method.

4.3.4.4.1 Closing a Document

When the user selects Close, the document shell closes the currently open document window.

If changes are to be saved when the document closes, the document shell follows the procedures described in the next section. If the user closes a document without saving changes to the current draft, the document shell releases the window state, leaving the document in the state it occupied at the last save.

4.3.4.4.2 Saving a Document

When the user selects Save, the draft object asks each part in the document to write itself to the document file. Each part writes its data and its embedded frames. The document shell then writes the window state to the document file.

When a document is saved, its data is saved in the context of the current draft. Only parts that have been modified since the creation of the current draft are asked to write themselves back into the document file. (Each draft of a document includes its own window state, allowing earlier drafts to be opened for viewing at any time.)

If the user specifies that a document is to be saved as stationery, the document shell sets a flag in the document file, to indicate that a document is stationery instead of an ordinary OpenDoc document.

4.3.4.5 Save a Copy

When the user selects Save a Copy, the document shell saves a copy of the current document into a new file.

4.3.4.5.1 Reverting a Document

When the user selects Revert To Saved, any changes that have been made since the last save are thrown away. The document shell releases the existing window state and restores the window state from the previously saved draft. As a result of reverting a draft, the active part and thus the selection and user-interface elements may change.

4.3.4.6 Draft History

When the user selects Draft History the document shell displays the draft history of the current document in a dialog box. The user can use the dialog to create new drafts of the document or to select previous drafts for editing.

4.3.4.7 Document Info

When the user selects Document Info, the document shell brings up a Document Info dialog box. This dialog box displays certain indicia about the document, and allows the user to change various document options.

4.3.4.8 Page Setup

When the user selects Page Setup, the command information is passed to the root part of the active window via its `handle_event` method.

4.3.4.9 Print

When the user selects Print, the command information is passed to the root part of the active window via its `handle_event` method.

4.3.5 Purging

OpenDoc includes a memory management mechanism intended to prevent failure due to exhaustion of memory resources. This mechanism is unrelated to the reference-counted object memory management mechanism discussed earlier.

Every subclass of `Object`, including subclasses of `Part`, must implement a `Purge` method. When the document shell calls a part's `Purge` method, that part should free the amount of memory requested in the `Purge` call. A part must only free memory that is recoverable; the intent is that a part trim its nonessential memory usage, not that it render itself inoperable.

A part that has failed to allocate needed memory can request the document shell to purge other parts by calling the `StorageSystem::need_space` method.

Note that the implementation of the Purge method may do nothing in implementations in environments where the operating system manages memory.

4.4 Services

This section describes classes and methods that part editor methods call.

4.4.1 Storage

OpenDoc uses a system of persistent (external) storage that is built upon the native file-storage facilities of each of the individual OpenDoc platforms. The same fundamental storage concepts and structures are used consistently for document storage, Clipboard transfer, drag-and-drop, and linking.

Several OpenDoc classes work together to implement the storage system. These classes are collectively known as a **container suite**. The heart of the container suite is the `StorageSystem` class. The `StorageSystem` object manages a set of `Container` objects. Each `Container` object manages a set of `Document` objects. Each `Document` object manages a set of `Draft` objects. Each `Draft` object manages a set of `StorageUnit` objects. Each `StorageUnit` object manages a set properties, which are named sets of values. The `StorageUnitCursor`, `StorageUnitRefIterator`, and `StorageUnitView` classes assist in the manipulation of `StorageUnit`s and their properties. Document shells work with the `StorageSystem` through `StorageUnit` objects in this functional hierarchy. Part editors work with the `StorageUnit` objects and those objects below it.

4.4.1.1 The Document Shell's Perspective

Document shells use `StorageSystem::create_container` to make `Container` objects. `Container` objects provide the interface between the internal view of OpenDoc storage and the external platform specific storage mechanisms such as a file system.

Containers hold **documents**. OpenDoc does not specify the relationship between documents and external storage; there is no requirement that each document be stored in its own file.

The `Document` object is responsible for the creation, deletion, and manipulation of drafts. A **draft** is a specific version of an OpenDoc document.

Drafts are document-wide, and are only required to store those parts of a document that have changed in each draft.

Drafts can have names; draft names are unique within a document. `Document`-object methods can access drafts by draft ID, by object reference, or by relative position in the document. Drafts have a specific set of properties, including creation date, modification date, and user name.

Any number of users can simultaneously edit a draft although only the most recent draft of a given document can be edited. (Drafts are not allowed to be changed if there are other drafts that are based on them). Implementations of OpenDoc are allowed to restrict editing so that only one user can edit a draft at a time.

Each `Draft` object maintains a list of changes from its predecessor draft. The `Draft` object is responsible for creating new `Frame` objects, counting references to existing frames, and reclaiming their storage when no longer referenced.

Each open draft has a set of **draft permissions** associated with it. They specify the class of read-write access that a part editor has to the draft. When a part initializes itself, it gets the permissions of its draft (by calling the draft's `get_permissions` method), and behaves accordingly. For example a part editor must not make changes to a part when its draft has been opened read-only.

4.4.1.2 The Part Editor's Perspective

A part editor is passed a reference to its main `StorageUnit` object when it is invoked, either by OpenDoc calling its `init_part` or its `init_part_from_storage` method.

A storage unit is a set of **properties,** or categorized groups of data streams. Each property consists of a number of streams called **values,** each of which has a named type. Thus there can be several properties (named categories) in a single storage unit, and several values (typed streams) in a single property.

Figure 37 is a simplified diagram of the organization of a storage unit that shows these relationships.

Figure 37 The organization of a storage unit

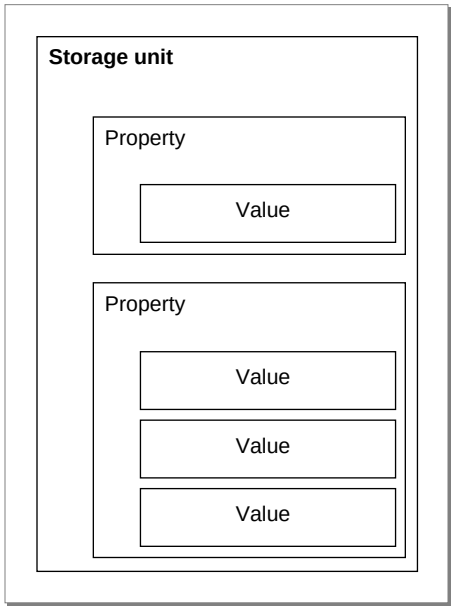
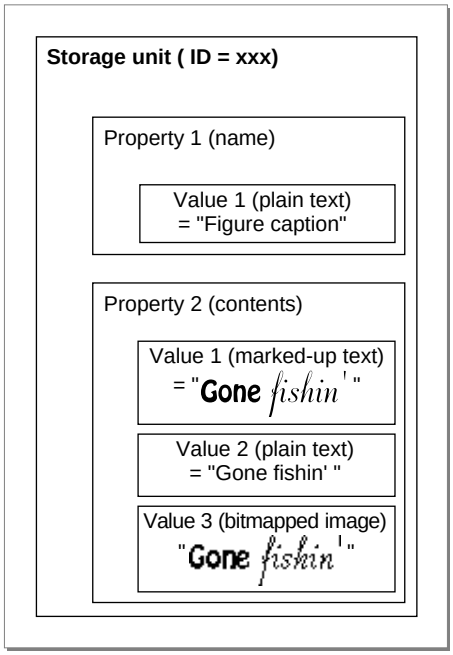


Figure 38 is a simplified diagram of a specific storage unit in use. The storage unit contains a figure caption. The part that owns this storage unit has created a name property, which names the item stored in this storage unit, as well as the contents property, which contains the primary data of this storage unit. This particular storage unit has one representation (value) of the name property, and three representations of the contents property (the text of the caption), as three different types of values: plain text, styled text, and a bitmap.

Figure 38 An example of a storage unit with several properties and values



Fundamental to the OpenDoc storage system is the convention that all values within a property are analogous, or equivalent, representations of the same data. Every value in the contents property in Figure 38, for example, is a complete representation of the item's contents, expressed in the type of data that the value holds.

Values in a storage unit can include references to other storage units, as described in the section. This means that storage units can be arranged in an ordered structure such as an acyclic graph, which is how OpenDoc stores the embedding hierarchy of a document.

Properties are added to a storage unit by `StorageUnit::add_property`. Values are assigned to properties by `StorageUnit::add_value`. The `add_value` method includes a type parameter which is used to distinguish among the many values that can be associated with a single property.

A **focus** mechanism exists that enables a part editor to select among the many properties contained in a storage unit and also to select among the many values contained in a property. A storage unit must be **focused** before data can be read from or written to it. Some methods, such as `add_value` and `add_property`, focus the storage unit as a side effect of their execution. A focus is explicitly obtained using `StorageUnit::focus`. Note that the existence of a focus prevents other parts from accessing the focused item.

The `StorageUnit` only maintains one focus. When a focus is created using a `StorageUnit` method it replaces any previously existing focus.

OpenDoc includes a mechanism, the `StorageUnitCursor` object, that allows foci to be easily saved and restored. A `StorageUnitCursor` is instantiated using `StorageUnit::create_cursor` and `StorageUnit::create_cursor_with_focus`. The focus is set to the one saved in a `StorageUnitCursor` with `StorageUnit::focus_with_cursor`.

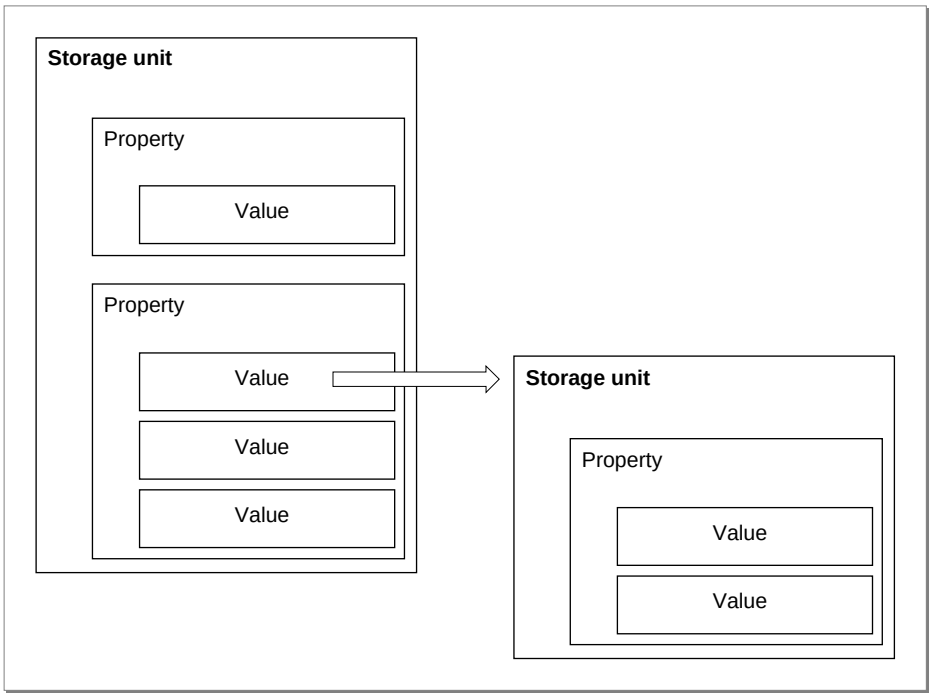
Even though the `StorageUnit` does not allow more than one focus at a time, OpenDoc make it possible for there to be more than one active focus via the `StorageUnitView` object. `StorageUnit::create_view` instantiates a `StorageUnitView` object that can be used for that purpose.

Methods of the `StorageUnit` and `StorageUnitView` allow the manipulation of value data. The size of the value data is obtained using the `get_size` method. `get_value` extracts a specified number of bytes from a value; `set_value` writes a specified number of bytes to a value; `delete_value` removes a specified number of bytes from a value. `set_offset` allows the getting and setting to start anywhere in the value data.

Properties and values are removed from a storage unit by focusing on the property or value, and calling `StorageUnit::remove`.

Properties can hold special value: a **persistent reference.** A persistent reference is a number stored in a value that refers to another storage unit in the same document. The reference is preserved across sessions; if a document is closed and then reopened at another time or even on another machine, the reference is still valid.

Figure 39 Persistent references in a storage unit



There are two kinds of persistent references, **strong persistent references** and **weak persistent references**. They are treated differently in cloning.

To **clone** an object is to copy it by value. Cloning is performed using methods of the `Draft` object. When cloning, copies are made of all objects referenced by strong persistent references, whereas objects referenced by weak persistent references are not copied.

Persistent references are created by focusing the storage unit on the value where the reference is to be stored, and then calling either the storage unit's `get_strong_storage_unit_ref` or `get_weak_storage_unit_ref` method, passing it an object reference to the storage unit that is to be referred to. Then store the returned reference in the focused value. Such a reference is then said to be *from* the value *to* the referenced storage unit.

Persistent reference values are opaque and have no meaning except when stored in the property value that was focused at the time that the persistent reference was obtained. The `StorageUnit` and `StorageUnitView` classes provide methods for manipulating them. In addition, the `StorageUnitRefIterator` class enables a part editor to access each persistent reference in a focused storage unit value.

4.4.2 Frames and Facets

4.4.3 Windows and Canvases

4.5 Part Editors

A part editor is created by subclassing the `Part` object, and implementing its methods. The method are grouped into categories that implement specific features, such as linking. It is not necessary to implement methods for features that are not provided by the part.

The `Part` methods are called by OpenDoc. The method code may call the methods of other OpenDoc objects in order to use various services. In many cases, the values returned by calls to service methods become invalid when the `Part` method returns control to OpenDoc. These cases are indicated in the method descriptions in the Class Reference.

4.5.1 Initialization

Either the `init_part` or `init_part_from_storage` method is called by OpenDoc immediately after a `Part` object is instantiated. `init_part` is called when a part is instantiated the first time; `init_part_from_storage` is called subsequently. These methods perform all part-specific initialization.

4.5.2 Part Information

OpenDoc includes the concept of **part information**. Part information is private part data that describes its state. Everything that a part editor needs to be able to create itself must be in the part information. Note that there is no requirement that the entire state be represented by value in the part information; the part information can contain references to other part-specific storage such as files.

OpenDoc calls the `read_part_info` method when opening a part. The `write_part_info` method is called when saving a part. The `clone_part_info` method is called when copying a part.

Part editors can handle specific kinds of data as discussed in section 4.3.2.1, “Part Kinds Stored in a Part”. OpenDoc calls the `externalize_kinds` method to find out what part kinds are supported by a part editor.

4.5.3 Reading and Writing Parts

This section discusses how parts read and write its own content data, as well as related OpenDoc objects, such as frames, that it uses.

A part editor must be able to read and write any of its parts. The entire content of a part need not be read into memory at once, nor written to storage at once. However, reading a part must get it into a state in which it can accept events and handle requests, and writing it must assure that changes made by the user are not lost.

A part editor should never change the part kind of any part it handles without explicit user action. Any translations which must be done to make a part readable, whether performed by OpenDoc or by a part editor, are first selected by the user.

4.5.3.1 Writing A Part to Storage

A part must be able to write itself (write its data to persistent storage) when instructed to do so. The caller calls a part’s override of the inherited `externalize` method. The part then writes itself by writing its data into its storage unit.

As a minimum, a part must write one property: `prop_contents`. The `prop_contents` property contains the part’s intrinsic data. Note that a part can write its data in multiple formats—that is, representing multiple part kinds—in the `prop_contents` property. Each format is a separate stream, defined as a separate value in the property, and each value must be a complete representation of the part’s data.

Note also that a part can add other properties and values to its storage unit, and it can access them for its own purposes. However, remember that the OpenDoc binding process looks only at the value types of `prop_contents` when reading the part.

The fundamental method that a part must implement for writing its data to storage is the `externalize` method. A simple `externalize` method might include these basic steps:

1. Examine the part to see if the content has been changed since the last write. If not, take no action.
2. Call the inherited `externalize` method, to make sure that the appropriate persistent-object information for the part (such as modification date and time) is written to storage.
3. Get a reference to the part’s main storage unit by calling the part’s inherited `get_storage_unit` method. Focus the storage unit on the contents property and the part kind of the data to be written. Write the data to the storage unit, using its `set_value` method.

4.5.3.2 Reading A Part From Storage

A part must be able to read itself (reconstruct itself in memory by reading its stored data). Whenever a document in which a part is visible is opened, or whenever a part is added to a document, the part must be read into memory. Note that OpenDoc parts should always be ready to work from an empty storage unit as well. The two fundamental methods that a part must implement for initializing itself are `init_part` and `init_part_from_storage`.

`init_part` is called only once in your part's lifetime, when it is first created and has no stored data to read. In the simplest case, take these steps:

1. Call the inherited `init_persistent_object` method. `init_persistent_object` initializes the information the part needs as a persistent object. (The part's classes' initialization methods should always call their superclasses' initialization methods.)
2. Add a contents property (type `prop_contents`) and a value (of the part kind) to the storage unit passed to the part. This prepares the part for eventual writing of its data.
3. Initialize any data pointers or size values that the part maintains, and mark the part as clean (unchanged).

`init_part_from_storage` is similar to `init_part`, except that it also reads in data. The caller of the part's `init_part_from_storage` method supplies the part with a storage unit, from which the part reads itself. Reading is basically the reverse procedure from writing. The part retrieves, from the `prop_contents` property, the value (specified by part kind) that represents the data stream to read. In the simplest case, take these steps:

1. Call the inherited `init_persistent_object_from_storage` method.
2. Focus the storage unit passed on its contents property and the value type of the part kind.
3. Get the size of the value, using the `get_size` method.
4. Read the data into the part's buffer, using the `get_value` method.

Reading in a part's data can involve reading in other storage units besides the part's main storage unit. In each case, a persistent reference within the main storage unit's `prop_contents` property leads to information stored elsewhere.

4.5.3.3 Creating Additional Storage Units

A part can use persistent references to create an auxiliary storage unit for keeping additional data. The auxiliary unit must be referenced from the part's main storage unit, using a strong persistent reference. In brief, follow these steps:

1. Call the `create_storage_unit` method of the part's draft to create the storage unit to hold the data.
2. Focus the part's main storage unit on the value in the contents property that is to contain the persistent reference. (It is not necessary to create a separate value to hold the reference.)
3. Create the persistent reference, by calling the `get_strong_storage_unit_ref` method of your main storage unit.
4. Store the reference anywhere in the value; the only requirement is to be able to recognize and retrieve the reference later. Write the value to persistent storage, by calling the `set_value` method of the main storage unit.

The part can later retrieve the auxiliary storage unit from the main storage unit, in this way:

1. Focus the main storage unit on the value that contains the persistent reference.
2. Read the value, using the `get_value` method of the main storage unit. Retrieve the persistent reference from the value's data.
3. Get the storage unit ID of the auxiliary storage unit from the persistent reference, by calling the `get_id_from_storage_unit_ref` method of the main storage unit.
4. Get the storage unit itself by passing its ID to the `get_storage_unit` method of the draft.

4.5.3.4 Storing and Retrieving Embedded Frames

A part does not explicitly store the data of embedded parts; their own part editors take care of that. A part does, however, store the frames embedded in it.

The process of storing an embedded frame for a part is simple. Store a strong persistent reference to the embedded frame object; OpenDoc takes care of storing the frame itself. As when creating and storing a reference to a storage unit (described in the previous section),

1. focus the storage unit on the contents property and the value that is to contain the persistent reference to the embedded frame
2. create the persistent reference and store it in the value
3. write the value to persistent storage

When instantiated at a later, time, the part can then retrieve the frame by

1. focusing the storage unit on the value containing the persistent reference
2. retrieving the reference
3. getting the storage unit ID of the frame
4. getting the frame itself

4.5.3.5 Removing an Embedded Part

To remove an embedded part from a part, take these steps:

- Remove whatever content structures that refer to the embedded part's frame.
- Remove the frame from the part's embedded-frames list (with which the part allows callers to iterate through its embedded frames
- Call the frames' `remove` method, to release the frame objects.

4.5.4 Binding

Part editors can support multiple part kinds, as discussed in section 4.5.4, "Binding". If a part editor has just been bound to a part whose previous editor used a different part kind, or if a part editor supports more than one part kind and the user has specified that the part is to be changed, the document shell calls the part's `change_kind` method, passing it the new part kind. The part editor must start manipulating the part's data in the new format, and store the part's data in a new order, with the new part kind as the highest-fidelity (first) data stream.

4.5.5 Part Wrappers

OpenDoc has the ability to replace a part editor for a part with a different part editor without having to close the part and reopen it with the new editor. This is accomplished through the use of part wrappers which hide the underlying `Part` objects so that they can be replaced without invalidating references.

Part editors normally work with a private subclass of the `Part` object known as a **part wrapper**. The `Part` object encapsulated by the part wrapper is known as the **real part**. OpenDoc only allows one part editor at a time to access the real part. A part editor can determine whether or not an object of the `Part` class is the real part using the `is_real_part` method. A part editor can get a reference to the real part using the `get_real_part` method; this also locks the real part for the calling part editor's exclusive access. The `release_real_part` method is used to relinquish that access.

Part editors do not implement these methods; they are available for a part editor to use.

4.5.6 Drawing

A part displays its content in response to a call to its `draw` method:

When it receives this call, a part editor draws the part content in the given facet. Only the portion within the update shape (defined by the `invalid_shape` parameter) needs to be drawn. The update shape is based on previous `invalidate` calls that were made involving this facet, or on updating needed because of part activation or window activation. The shape is expressed in the coordinate system of the facet's frame.

There are several steps that a part needs to take to draw itself:

1. To decide how to display itself, a part examines the following information in the given facet and its frame:
    - The view type of the frame tells the part whether it is to display itself with an iconic or framed representation. The part displays itself in the specified frame according to the frame's view type.
    - The presentation of the frame tells the part what kind of presentation the part content is to have, if its view type is framed. If a part does not support the frame's presentation, it call the frame's `set_presentation` method to substitute a correct value.
    - A part may be within a selection in its containing part, in which case it may need to be highlighted appropriately for the selection model of the containing part. Call the `get_highlight` method of the facet to see whether the content should be drawn with full highlighting, dim (background) highlighting, or no highlighting.
    - Check to see if the part needs to draw borders around any link sources and destinations in its content.
2. The part also examines the canvas that it is being imaged on. The part calls the `get_canvas` method of the facet to get a reference to the drawing canvas.
    - If the canvas is dynamic, the result of the canvas's `is_dynamic` method is `TRUE`.
    - If the canvas is offscreen, the result of the canvas's `is_offscreen` method is `TRUE`.
3. The part must make sure that any required platform-specific graphics structures are prepared for drawing, and

that the drawing context for the facet is set correctly. A part can then draw its contents, using platform-specific drawing commands.

4.5.6.1 Invalidating and Validating Part Content

To mark areas of a part's content that need redrawing, modify the update shape of the canvas on which the part is imaged by calling the `invalidate` method of the display frames or their facets, passing a shape (`invalid_shape`), expressed in the frames own coordinates, that represents the area that needs to be redrawn. OpenDoc adds that shape to the canvas's update shape, and when redrawing occurs once again, that area is included in the `invalid_shape` passed to the part's `draw` method.

Likewise, to remove previously invalid areas from the update shape of the canvas, call the `validate` method of the frame or facet, passing a shape (`valid_shape`) that represents the area that no longer needs to be redrawn.

4.5.6.2 Drawing When a Window is Scrolled

When the entire contents of a window is scrolled, the scrolled position of the contents is determined by the value of the internal transform of the root part's frame. For any of its embedded frames that are made newly visible or newly invisible by scrolling, the root part should create or destroy their facets by calling the root facet's `create_embedded_facet` or `remove_facet` method.

To force redrawing of those parts of the window that need to be redrawn after scrolling, the root part marks the area that needs to be redrawn as invalid, by calling the root facet's `invalidate` method, so that it will be redrawn when an update event occurs.

4.5.6.3 Redrawing a Frame When it Changes

If a containing part changes an embedded frame's shape, the frame's part (or the containing part) may need to refresh its presentation. Take these steps:

1. Change the frame's shape as appropriate.
2. Change the clip shape—and external transform, if necessary—of the frame's facet to correspond to the new frame shape.
3. Call the embedded part 's `frame_shape_changed` method, passing it the new frame shape.
4. If there are undrawn parts outside of the new facet, invalidate the portions of the part's intrinsic content that correspond to the difference.

When the next update event is generated, the `draw` methods of the embedded part and embedding part, as appropriate, are called to draw the invalidated areas.

4.5.6.4 Drawing When a Part is Scrolled

When a user scrolls the contents of an embedded part, the containing part takes no role in the repositioning or redrawing. The embedded part sets its frame's internal transform to the appropriate value and calls the facet's `invalidate` method to force the redraw.

Parts whose contents are to be scrolled, take these steps:

1. The frame's `set_internal_transform` method, is called to set the transform's offset values appropriately. How the offset values that are passed to `set_internal_transform` are obtained depends on what kinds of events are interpreted as scrolling commands, and how the part handles them.
2. If any embedded frames become visible or invisible as a result of the scrolling, create or delete facets for them.
3. Call the facet's `invalidate` method to force a redraw at the next update event. the frame's image by the scrolled amount, and only invalidate the portion of the facet that represents part content scrolled *into* view.

Drawing the contents of frames that include scroll bars is more complicated, because the content must be clipped so that it won't draw over the scroll bars, and because the content can be in a scrolled position but the scroll bars themselves should not move.

4.5.6.5 Drawing Selected Parts

A part determines how selecting works and how selections are drawn inside parts. However, if a part editor supports embedding of other parts, it should support a consistent selection behavior and appearance.

To show that a part embedded within a containing part is selected, give it an appearance that depends on the content's selection model, and on whether or not the part is selected by itself or as part of a larger selection that includes the part's intrinsic content:

To select an embedded part alone when it is viewed in a frame, the user can perform several actions, such as activating the part and then clicking on its frame border, or using a lasso tool or other selection method supported by the containing part. Note that when a frame is selected, its part is not active; the menus displayed are those of the containing part.)

If the framed part alone is selected, a containing part draws the frame border with a standard appearance—typically a gray line, corresponding to the frame shape or the used shape, with resize handles if the user is permitted to resize the frame.

To select an embedded part alone when it is viewed as an icon, the user places the mouse pointer over a part's icon (that is, anywhere within the active shape of the facet displaying the icon) and clicks the mouse button. Because the view type of the facet is iconic rather than framed, OpenDoc sends the mouse-up event to the part (as an event type of `evt_mouse_up_embedded`).

The part, in this case, does not draw the frame border of the embedded part at all, instead notifies the part that it is highlighted by calling the `change_highlight` method of the embedded part's facet, specifying a highlight value of `full_highlight`. It is then up to the embedded part to draw its highlighted appearance, either by using a highlight color or displaying the selected version of its icon.

- The appearance of an embedded part that is enclosed within a range of a part's selected intrinsic content should be analogous to the appearance of the intrinsic content itself. Draw the frame border with a selected appearance, if appropriate, and call the `change_highlight` method of the embedded part's facet, so the part will know how it should highlight itself:
    - If a part highlights selections with a highlight color or inverse video, do not draw frame borders around embedded parts within the selection, set their facets' highlight value to `full_highlight`.
    - If a part highlights selections by drawing frame borders around individual objects, draw frame borders with a selected appearance around any embedded parts, and set their facets' highlight value to `no_highlight`.
    - If a part highlights selections by drawing a dynamic marquee or lasso border around the exact area of the selection, do not draw frame borders around embedded parts, set their facets' highlight value to `no_highlight`.

A part should allow for multiple selection, so the user can select more than one frame at a time; it should also support Select All on its own contents; and it should furthermore allow for selection of "hot" parts—parts such as controls that normally perform an action (like running a script) on a single click—without executing the action.

4.5.6.6 Drawing with Scroll Bars

This section discusses two methods for providing scroll bars for embedded parts. Note that these approaches work not only for scroll bars, but for any non-scrolling adornments associated with a frame.

4.5.6.6.1 Placing Scroll Bars Within A Frame

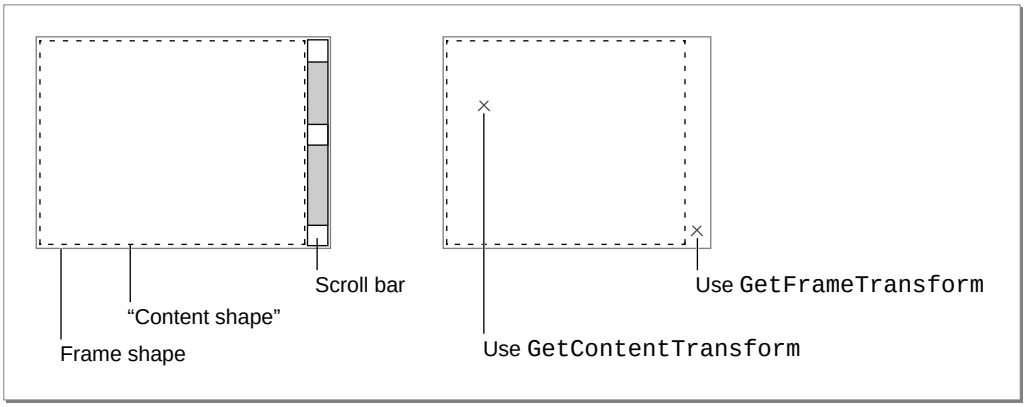
If a part creates scroll bars for its content inside the margins of its frame, remember that the frame shape includes the areas of the scroll bars. To draw properly, the part needs to account for the fact that the content can scroll but the scroll bars themselves should remain stationary.

Note that the method discussed in this section applies equally well to a root part that needs to put scroll bars in its window.

4.5.6.6.1.1 Clipping and Scrolling a Part's Content

One approach is to create an extra, private "content shape," as shown in Figure 40. The content shape is the same as the frame shape, except that it does not include the areas of the scroll bars.

Figure 40 Using a “content shape” within a frame shape for drawing



Content shape can be defined in terms of frame coordinates, and can be stored anywhere, although the frame’s part info field is a reasonable place.

When drawing a part’s contents—the portion within the area of the content shape— take the current scrolled position of the content into account. Include the frame’s internal transform by using the transform returned by `get_content_transform` to set the origin, and use the content shape as the facet’s clip shape when drawing. When drawing the scroll bars, however, ignore the current scrolled position of the content. Use the transform returned by `get_frame_transform` to set the origin, and use the frame shape as the facet’s clip shape.

4.5.6.6.1.2 Clipping Embedded Frames

One complication of this method is that OpenDoc knows nothing of the content shape, and therefore cannot clip the facets of embedded frames to it. OpenDoc clips all embedded facets to the area of the frame’s facet, but that area includes the scroll bars as well; thus embedded parts can draw over the area of the scroll bars.

To avoid this problem, intersect the clip shapes of each embedded facet with the content shape before the facet draws itself.

4.5.6.6.2 Placing Scroll Bars in a Separate Frame

A part can place scroll bars or adornments in completely separate frames from its content, to avoid having to define a separate, private “content shape.” This method, however, requires the overhead of defining more objects.

A part can request one display frame that encloses both the scroll bars and the content. This frame has an internal transform of identity: it does not scroll. Draw the scroll bars directly in this frame.

Request a second display frame that can scroll and in which the part draws its content. Place the second frame within the first; that is, make the nonscrolling frame the containing frame of the scrolling frame. Because both frames are display frames of the same part, call the `set_subframe` method of the second frame to make sure OpenDoc knows that it is a subframe of its containing frame, and draws the active frame border in the correct location.

The containing part, as usual, must make separate facets for both frames. Then, when a mouse even occurs in the non-scrolling frame, the part can interpret it and set the internal transform of the scrolling frame accordingly.

With this approach, a part does not need to take any special care to manage clip shapes of any embedded facets.

4.5.6.7 Drawing Directly to the Window

If a part editor needs to draw interactively, such as for rubber-banding or for sweeping out a selection while the mouse button is held down to provide user feedback, it can draw directly on the root facet’s canvas (the window canvas).

OpenDoc provides `Facet` methods equivalent to those used for drawing to the part's own canvas, that can be used to draw directly to the window:

- The `acquire_window_content_transform` method, equivalent to `get_content_transform`, converts from the part's content coordinate space to window-canvas coordinate space.
- The `acquire_window_frame_transform` method, equivalent to `get_frame_transform`, converts from the part's frame coordinate space to window-canvas coordinate space.
- The `acquire_window_aggregate_clip_shape` method, equivalent to `get_aggregate_clip_shape`, converts the part's facet clip shape to a clip shape in window-canvas coordinate space.

Use the values returned by these methods to set up the part's drawing structure when drawing directly to the window canvas. Note that, in cases where there is no canvas in the facet hierarchy other than the window canvas, the results returned by each of these pairs of methods are the same.

4.5.6.8 Asynchronous Drawing

A part editor may need to display its part asynchronously, rather than in response to a call to its `draw` method. For example, a part that displays a clock may need to redraw the clock face exactly once every second, regardless of whether or not an update event has resulted in a call to redraw. Also, visual feedback that provided to a user dragging the part's content is drawn asynchronously. Asynchronous drawing is very similar to synchronous drawing, except for these minor modifications:

1. Determine which of the part's frames should be drawn. The part can have multiple display frames, and more than one may need updating. Because the part stores its display frames privately, only the part can determine which frames need to be drawn in.
2. For each frame being displayed, the part must draw all facets. The frame's `create_frame_facet_iterator` method returns an iterator with which the part can access all the facets of the frame. (Alternatively, a part can keep a private list of facets.) Draw the part's contents in each of these facets, using the steps listed for synchronous drawing.
3. After drawing in a facet, call its `drawn_in` method to tell it that it has been drawn in it asynchronously. If the facet is on an offscreen canvas, calling `drawn_in` lets it get copied into the window, because the facet then calls the `invalidate` method of its canvas, which in turn calls its owning part's `canvas_changed` method.

4.5.6.9 Offscreen Drawing

There are several situations in which a part may want to create an offscreen canvas. For example, it may want to increase performance with double-buffering, drawing complex images offscreen before transferring them rapidly to the screen. Alternatively, it may want to perform sophisticated image manipulation, such as drawing with transparency, tinting, or using complex transfer modes.

Parts are likely to create offscreen canvases for their own facets for double-buffering, in order to draw with more efficiency and quality. Parts are likely to create offscreen canvases for their embedded parts' facets for graphic manipulation, in order to modify the imaging output of the embedded part and perhaps combine it with their own output.

4.5.6.9.1 Adding and Removing Canvases

To create a canvas and attach it to a facet, take these steps:

1. Create and initialize the platform-specific drawing structure that underlies the canvas.
2. Create a canvas object, using the `create_canvas` factory method of `Facet`. Calling this method assigns the platform-specific drawing structure to the new canvas, and also defines the canvas as static or dynamic, and onscreen or offscreen.
3. Assign the part as owner of the canvas by calling the `set_owner` method of the canvas.
4. Assign the canvas to a facet. Because only the owner of a canvas can remove it from a facet, the timing of assigning the canvas is important:
    - If the part is a containing part assigning an offscreen canvas to one of its embedded parts, assign the canvas when the embedded facet is first created—that is, when the first call to the `create_embedded_facet`

method of the embedded part's containing facet is made. Otherwise, the embedded part may assign a canvas to the facet, precluding the embedding part from doing so.

- If a part is a containing part and one of its embedded parts has an existing facet with no assigned canvas, add one by calling the `set_canvas` method of the embedded facet. When this is done, OpenDoc communicates the change to all embedded parts that use that facet, by calling their `canvas_changed` methods.
- A part can add a canvas to any of its own display frames' facets at any time, as long as the facet does not already have an assigned canvas. It is probably best to attach the canvas as soon as the facet is created, by calling `set_canvas` from within the part's `facet_added` method. If the containing part has already attached a canvas to the new facet, a different canvas cannot be assigned to it.
- If a part absolutely needs to attach its own canvas to a facet that already has an assigned canvas, it can get around this restriction by creating a subframe of the facet's frame, creating a facet for that frame, and assigning the canvas to that facet.

To remove a canvas from a facet, take these steps:

1. Call the facet's `set_canvas` method, passing it a null value for the canvas reference.
2. Delete the graphics-system-specific structures that had been referenced by the canvas.
3. Delete the canvas object.

4.5.6.9.2 Drawing to an Offscreen Canvas

Drawing on an offscreen canvas is essentially the same as drawing on an onscreen canvas. Set up the environment as usual: obtain the canvas and its pointer to the system-specific drawing structure as usual, and use the facet's transform and clip information as usual.

If a part draws asynchronously on a facet's offscreen canvas, call the facet's `drawn_in` method—to assure proper updating of the offscreen canvas to the window canvas. If a part's facet is moved to an offscreen canvas while it is running, OpenDoc calls the part's `canvas_changed` method, so that the part can know that it must call `drawn_in` if it is drawing asynchronously.

Perform invalidation the same way for onscreen and offscreen canvases. Call the `invalidate` or `validate` methods of the display frames and their facets, to accumulate the invalid area in the update shape of the offscreen canvas.

To bypass the offscreen canvas and draw directly on the window canvas, obtain the window canvas when setting up the environment, and use the appropriate calls (such as `acquire_window_content_transform` instead of `acquire_content_transform`) when setting the origin and clip.

4.5.6.9.3 Updating an Offscreen Canvas

The part that owns an offscreen canvas is responsible for transferring its contents to the parent canvas. Only the part that created the canvas can be assumed to know how and when to transfer its contents. The canvas may have a different format from its parent (one may be a bitmap and the other a display list, for example); the owner may want to transform the contents of the canvas (such as by rotation or tinting) as it transfers; or the owner may want to accumulate multiple drawing actions before transferring.

When a containing part has placed one of its embedded part's facets on an offscreen canvas, it should force the embedded part to draw before the containing part itself draws any of its own contents. This ensures that the contents of the offscreen canvas are up to date, and can safely be combined with the containing part's contents when the containing part draws.

If an embedded part displays asynchronously and uses `invalidate` or `validate` calls to modify the update shape of its offscreen canvas, the offscreen canvas calls its owning part's `canvas_updated` method to notify it of the change. The owning part can then transfer the updated content immediately to the parent canvas, or it can defer the transfer until a later time, for efficiency or for other reasons.

4.5.6.10 Drawing With Multiple Frames or Facets

OpenDoc has built-in support for multiple display frames for every part, and multiple facets for every frame. This arrangement gives great flexibility in designing the presentation of a part and of parts embedded within a part. This section summarizes a few common reasons for implementing multiple frames or multiple facets.

4.5.6.10.1 Multiple Frames

Several uses for multiple frames have already been noted, most of which involve the need for simultaneous display of different aspects or portions of a part's content:

- A part whose frame is opened into a part window requires a second frame as the root frame of that part window.
- A part that flows text from one frame to another naturally needs more than one frame.
- Parts that allow for multiple different presentations of their content are likely to display each kind of presentation (top view vs. side view, wire-frame vs. rendered, and so on) in a separate frame.
- Parts that display scroll bars, adornments, or palettes in addition to their own content might use separate frames for such items.

In general, it is the part displayed in a set of frames that initiates the use of multiple frames. The containing part does, however, have ultimate control over the number and sizes of frames for its embedded parts.

4.5.6.10.2 Multiple Facets

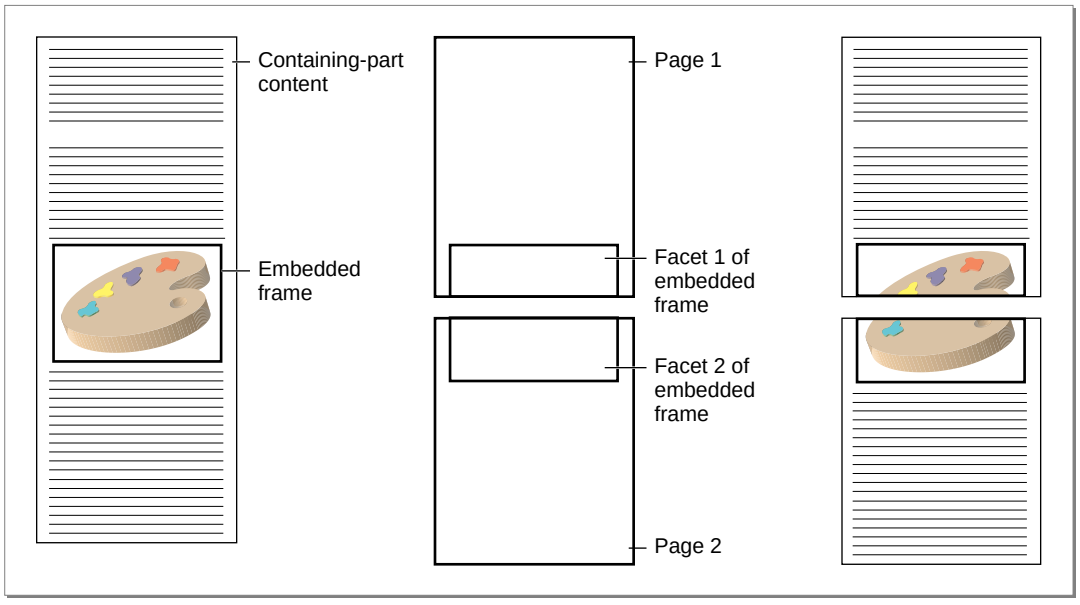
The use of multiple facets allows a containing part to duplicate, distort, and reposition the content displayed in its embedded frames. Normally, even with multiple facets, all of the facets of a frame display within the area of the frame shape, because the frame represents the basic space agreement between the embedded part and the containing part. However, the containing part always controls the facets to be applied to its embedded frames, and so the containing part can display the facets in any locations it wishes.

4.5.6.10.2.1 Drawing an Embedded Frame Across two Display Frames

Perhaps the most common use of multiple facets for a frame may be when an embedded frame spans the boundary between two containing frames.

In Figure 41, for example, a word-processing part uses a display frame for each of its pages. An embedded part's display frame spans the boundary between page 1 and page 2. The word-processing part then defines two facets for its embedded frame: one embedded in the facet for page 1, and the other embedded in the facet for page 2.

Figure 41 Drawing an embedded frame that spans two display frames

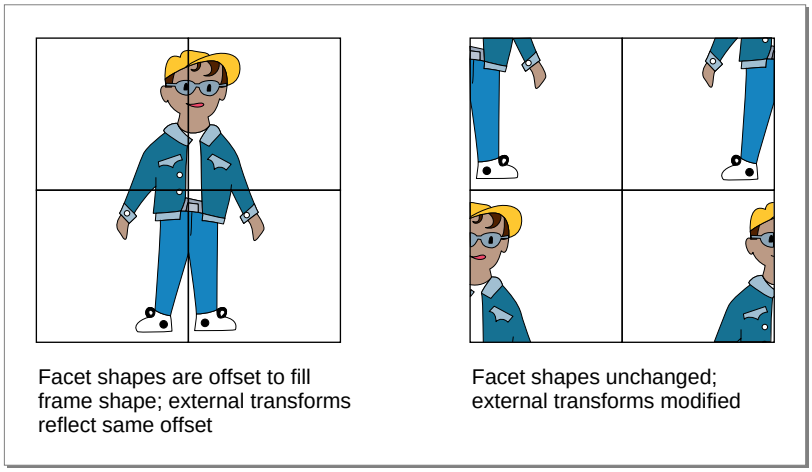


4.5.6.10.2.2 Multiple Views of an Embedded Frame

One simple use of multiple facets is for a containing part to provide a modified view (such as a magnification) in addition to the standard view of an embedded frame. Another effect achievable with multiple facets is to tile the area of an embedded frame with multiple miniature images of the frame's contents.

A similar, slightly more complex example is shown in Figure 42. In this case, multiple facets are used to break up the image in an embedded frame, to allow for moving the tiled parts around like a sliding puzzle.

Figure 42 Using multiple facets to rearrange portions of an embedded image

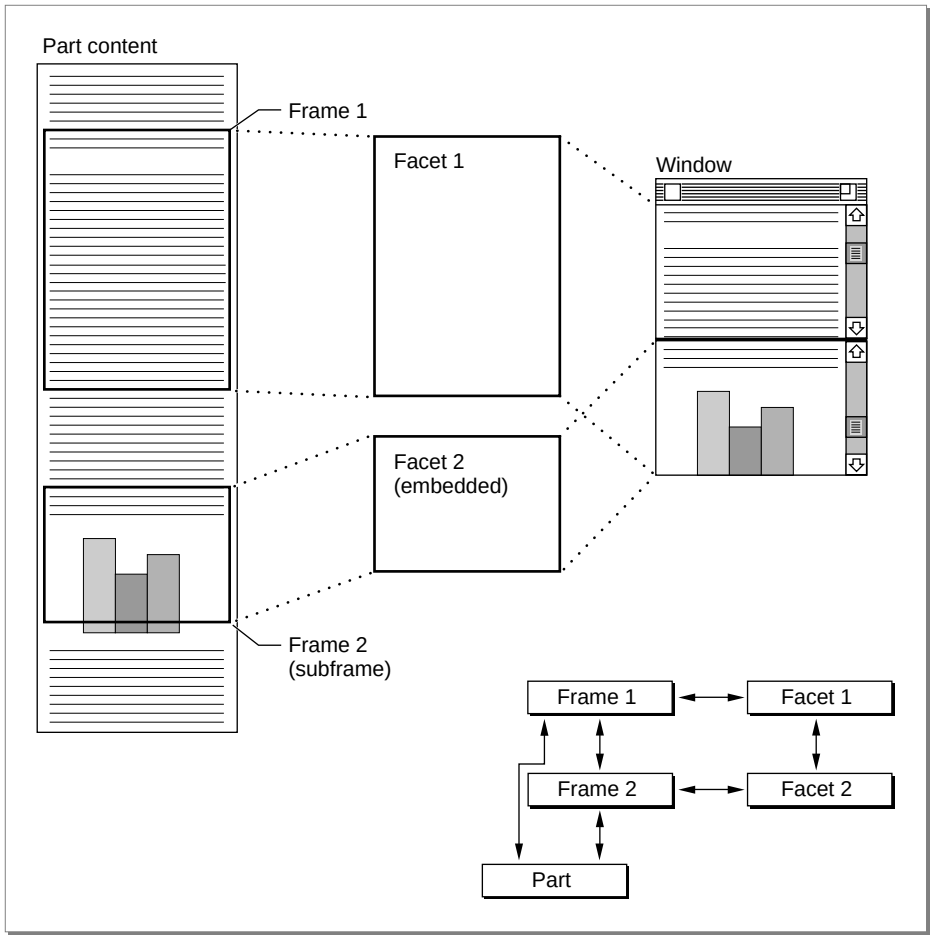


The clip shapes for the facets in Figure 42 are smaller than the frame shape and have offset origins so that they cover different portions of the image in the frame. Initially, as shown on the left, their external transforms have offsets that reflect the offsets in the shape definitions, and the facets just fill the area of the frame. Subsequently, as shown on the right, the containing part can cause any pair of facets to change places in the frame simply by swapping their external transforms.

4.5.6.10.2.3 Providing Split-Frame Views

One of the most useful implementations of multiple facets may be for constructing **split-frame views,** in which the user can independently view and scroll two or more portions of a single document. This use of multiple facets is somewhat more complex than the others presented here, because it also involves use of **subframes,** embedded frames that display the same part as their containing frame. Figure 43 shows one example of how to implement a split-frame view with a single subframe.

Figure 43 Using subframes to implement a split-frame view



In Figure 43, the part to be displayed has one frame (frame 1) and facet (facet 1) that represent its current overall frame size and display. The part creates frame 2 as an embedded frame, specifying that the containing frame is frame 1. The part then calls the `is_subframe` method of frame 2, causing frame 2 to be also a display frame of the part. (The diagram in the lower right of Figure 43 shows the object relationships.)

The part now needs to negotiate for frame 2 only with itself. It creates a facet (facet 2) for frame 2, embedded within facet 1. The part can now position frame 1 and frame 2 independently, by changing their internal transforms. It uses the external transform to place facet 2 within facet 1 as shown, and the two facets together fill the area of frame 1 and show different parts of the document.

4.5.6.11 Providing Cached Presentations

If a document is saved and then is subsequently opened on another machine on which the part editor(s) that created the document are not installed, and neither is any part editor or viewer that can read or translate the saved part kind, the OpenDoc editor of last resort opens the parts and displays their frames as gray boxes.

A part can ensure that any user of any document on any machine can display at least a static representation of the contents of each of its frames if, when writing out your part, the editor includes—as one of its part kinds—bitmap images of each of its frames, in a standard public format for the platform. In that case, there should always be at least one editor that can read the standard format and display a basic representation of the visible portions of your part's content.

4.5.6.12 Invalidating and Updating

When a portion of a window needs to be redrawn because it has been invalidated by actions taken by the documents' parts or by the system (as when a window is uncovered), the document shell receives notification of that fact and

passes an update event to the dispatcher, which calls the `update` method of the window; the window in turn calls the `update` method its root facet.

The root facet's `update` method examines all of the root facet's embedded facets (and their embedded facets, and so on), and marks each one that intersects the area that must be updated. Then, each facet that needs to be redrawn calls the `draw` method of its frame's part, passing it the appropriate shape that represents its portion of the area to be updated.

4.5.7 Printing

To print a document, the user selects Print from the Document menu, and the root part displays a print dialog box. If the user confirms the print command, the root part then sets up the environment for printing and prints the document.

In printing, the printer is represented by a separate canvas object from the screen display, although part editors in general use normal drawing calls to print their content.

Because any part can be the root part of a document, all parts should in general support the printing commands. The root part has greater responsibility in printing, but all visible embedded parts are asked to draw themselves, and they can adjust their drawing process for the presence of a static canvas. They can also directly access the platform-specific printing structure, if necessary.

When a part is the root part, it also handles the Page Setup command.

OpenDoc does not replace a platform's printing interface; OpenDoc objects provide access to platform-specific structures and provide a context within which parts make platform-specific calls.

This section first discusses the specific printing responsibilities of the root part, then those of embedded parts, and finally notes aspects of printing that apply to all parts.

4.5.7.1 Root-Part Responsibilities

If a part is the root part, it defines the basic printing behavior of its document. It sets things up, it makes the platform-specific printing calls, and it performs any frame negotiation that is to occur. It also controls whether or not embedded parts that use a different imaging or printing system can be printed.

4.5.7.1.1 Printing the Document

Once the user has confirmed a print dialog, a part is ready to print. If it supports layout negotiation with its embedded frames for printing, it creates a separate printing frame, as discussed in the next section, "Performing Frame Negotiation During Printing." Otherwise, it uses its display frame (which is the root frame of its window) as the printing frame. Then it takes these general steps:

1. Create a platform-specific printing structure.
2. Create an external transform and clip shape for the printing facet. Make the clip shape equal to the part's page size.
3. Create a new facet for the printing frame, using the Window state object's `create_facet` method. Assign the transform and clip to the printing facet.
4. Create a static canvas with the printing facet's `create_canvas` method, and assign the printing structure to it with the `set_platform_print_job` method. Also, use the `set_platform_canvas` method to assign the drawing structure associated with the printing structure to the canvas.
5. Assign the canvas to the printing facet by calling the facet's `set_canvas` method. Notify the printing frame of the presence of the printing facet by calling its `facet_added` method.
6. Loop through all pages in the part's content area, and print. For each page, do the following:
    - Reset the printing facet's clip shape and/or transform so that it covers the next page.
    - Create facets for the newly visible embedded frames, and release facets for frames no longer in view. At this point, allow frame negotiation with embedded parts, if supported (see "Performing Frame Negotiation During Printing," next).
    - Invalidate the printing facet's area and force a redrawing, by calling the facet's `update` method.
7. When finished printing, clean up. Remove the printing facet from the printing frame and release it. Delete the platform-specific structures that were created. If the part created a printing frame, remove it also.

4.5.7.1.2 Performing Frame Negotiation During Printing

If a part's document uses an identical page layout for printing as for screen display, printing is simpler. Just use the part's existing display frame (the window's root frame) as the frame for printing. That way, it is not necessary to create any new frames (including embedded frames) or to engage in frame negotiation with embedded parts.

However, a part may want to allow the printed version of a document to have a somewhat different layout from the onscreen version. For example, it may want to give embedded parts a chance to remove scroll bars and resize their frames, or otherwise lay themselves out differently when printed. To do that, take these steps when preparing for printing:

1. Create a new, nonpersistent frame as a display frame for the part, to be used as the printing frame.
2. Create new frames for each of the embedded parts, with the printing frame as their containing frame.

Then, during printing, when creating the facets for the newly visible embedded frames on each page that to image for printing, permit frame negotiation with embedded parts. That negotiation can result in changes in the sizes of the frames and their facets. Then continue with the printing process for that page.

4.5.7.2 Embedded-Part Responsibilities

To make sure that a part handles printing properly, when a part is an embedded part and is assigned a new display frame or facet, it examines the associated canvas to determine whether it is dynamic or static, and performs frame negotiation accordingly. On a static canvas, if the part is an embedded part, it might engage in a frame negotiation to make all of its content visible, or it might resize the frame to account for elimination of scroll bars or other items that are appropriate for screen display only.

4.5.7.3 Issues for All Parts

A part draws to the printer canvas just as it draws to its window canvas. The part's `draw` method is called when the facet that displays the part is being imaged. Set up the environment for drawing as usual, by obtaining the platform-specific drawing structure from the facet's canvas (using its `get_platform_canvas` method).

When a part draws to a static canvas (use the `is_dynamic` method of the canvas to find out), it may not want to draw adornments and interactive features such as scroll bars that are appropriate for screen display only.

On a static canvas a part may also want to perform color matching to take into account the characteristics of the device on which you are drawing.

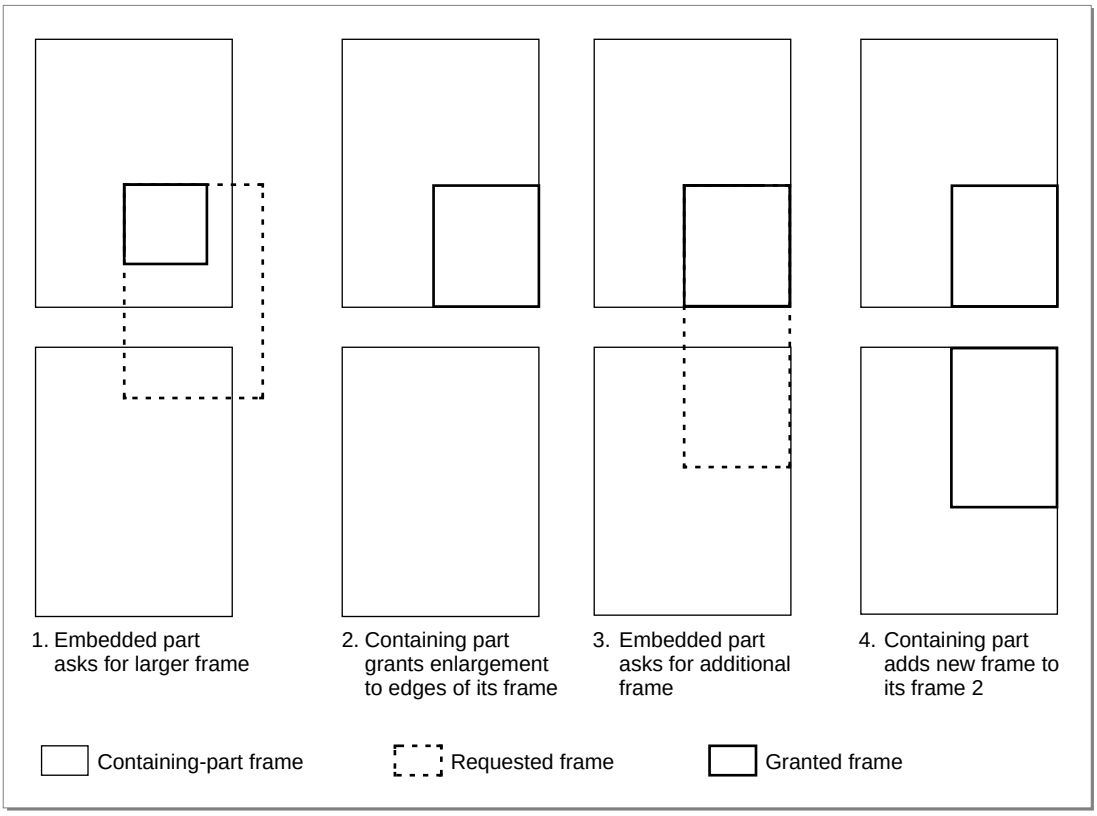
Note that all printing canvases are static, but not all static canvases are for printing. The presence of a static canvas does not guarantee that a print job or job object exists. Call the `has_platform_print_job` method of the canvas to find out if a printing structure exists. If it does, it is available through the `get_platform_print_job` method of the printing canvas. A part may want to access the print job or job object directly to determine, for example, whether it is printing on a PostScript printer.

4.5.8 Frame Negotiation

Each part in an OpenDoc document controls the positions, sizes, and shapes of the frames embedded within it. At the same time, embedded parts may have reasons to want to change the sizes, shapes, or numbers of frames they are displayed in. This section explains both how a part negotiates its display frame sizes with its containing part, and how it negotiates its embedded frames' sizes with embedded parts.

Frame negotiation is the process whereby an embedded part and its containing part agree on the what frame or frames the embedded part can have. Either party can initiate the negotiation, although the containing part has unilateral control over the outcome. Figure 44 shows an example of frame negotiation, in which an embedded part with a single display frame requests a larger frame size from its containing part, which has two display frames.

Figure 44 An example of frame negotiation with multiple frames



In this case, frame negotiation is initiated by the embedded part, whose frame is wholly contained within frame 1 of the containing part:

1. The embedded part asks the containing part for a significantly larger frame,. The embedded frame is not concerned with, and does not even know, where or how the larger frame will fit into the containing part's content.
2. The containing part decides, on the basis of its own content model, that the requested frame is too large to fit within frame 1. The containing part instead increases the size of the embedded frame as much as it can, assigns it a place in its content area, and returns the resulting frame to the embedded part.
3. The embedded part accepts the frame given to it. If it were to repeat step 1 and ask for the original larger frame again, the containing part would simply repeat step 2 and send the same result frame back.
   But the embedded part still wants more area for its display, so it tries a different tack; it requests another display frame, this time to be embedded in frame 2 of the embedded part.
4. The containing part decides that the requested frame size will fit in frame 2. It assigns the frame a place within frame 2, and returns the frame to the embedded part.

4.5.9 Display Frame Manipulation

A part's **display frames** are the frames within which its contents are drawn. Parts do not directly create their own display frames; the part's containing part does that. When a part is first created or invoked, its containing part will have already provided at least one frame for it to display itself in.

4.5.9.1 Facets

When a part has more than one display frame, it must maintain information so that it knows how to display itself in each. It is the part's responsibility to update each frame that has been affected by any change to the part.

When a part's containing part has added a facet to one of the part's display frames, the display frame notifies the part of the addition by calling the part's `facet_added` method:

The part's `facet_added` method must perform certain actions to handle the addition of the new facet to one of its frames. Some actions depend on the nature and implementation of the part itself, but others are standard:

- The `facet_added` method creates facets for all embedded frames that are visible within the area of the added facet.
- The `facet_added` method stores any part info that it needs in the facet that is being added.
- The `facet_added` method examines the facet's canvas to make sure the part editor understands how to draw on that canvas, and return an error if it does not.

When a containing part removes a facet from one of a part's display frames, the frame notifies the part by calling the part's `facet_removed` method:

The part's `facet_removed` method must perform certain actions to handle the removal of the facet. In general, the actions performed by this method reverse those performed by `facet_added`. Typically, the method at least

- removes the facets for all embedded frames that were visible in the area of the removed facet
- deletes any part info data that was referenced in the facet

If a part is the root part in a window, it receives a `facet_added` call from the root frame when the window is opened, and a `facet_removed` call from the root frame when the window is closed.

4.5.9.2 Frame Size Negotiation

Because of editing operations in a part, or because of an undesirable frame size imposed by a part's containing part, a part may desire a change in the size or shape of its display frame. It must negotiate this change with the containing part.

The part starts by requesting a new frame size from its containing part. Depending on its current contents and other constraints such as gridding, the containing part may grant the requested size, return a different size, or in essence refuse the request by returning a size identical to the current frame size.

A part can request a new frame size by taking these steps:

- Call the `request_frame_shape` method of the part's display frame. The frame in turn calls the containing part's `request_frame_shape` method to forward the request.
- The containing part may honor the request, or it may decide on a different (usually smaller) shape. It returns the shape it will let the part's display frame have. The frame stores it as its frame shape, and returns that information.
- Use the returned frame shape to update the used shape for that frame. The part can also at this time update the active shape for its frame's facet.
- If the part does not wish to accept the new shape, it can call the frame's `request_frame_shape` method again, but with a different shape to avoid endless repetition of these steps. Alternatively, it can request an additional frame.

4.5.9.3 Additional Frames

A part may need a display frame in addition to the frame or frames already created by the containing part. For example, it may need an extra frame to flow content into, such as an additional column or additional page of text, or it may need a new frame to display the part with a new presentation.

To request another frame, the `request_embedded_frame` method of the containing part is called. One of the current display frames must be specified as a base frame; the new frame will be a **sibling** of the base frame, meaning that it is embedded at the same level as the base frame. Also, the new frame will be in the same group as the base frame. Additional information, such as the view type and presentation for the new frame, and whether it should **overlay,** or float over, the content of the containing part, is also passed.

When a new frame is created, OpenDoc automatically connects it to its part as one of its display frames. This ensures that frames are always valid and usable; the object that creates the new frame need do nothing beyond creating the frame itself.

To achieve that automatic connection, the part being displayed within the frame must respond to the `frame_added` method call that tells it that it has a new frame.

There are several things a part editor's `display_frame_added` method must do. Most of them depend on the nature and implementation of the part itself; however, here are some general actions it should take:

- It adds the new display frame to the part's internal list of display frames.
- It validates the view type and presentation of the new frame. The part should accept any view types that are in the required set of view types, plus any other view types or presentations that are supported. The `display_frame_added` method corrects these values if necessary.It adds any needed part info to the frame.

A part does not perform any layout or imaging tasks as a result of a display frame being added; it waits until its `facet_added` method is called, at which time the frame has become visible.

If a part is the root part in a document, its frame may want to become active when its window becomes active. To make sure that happens, the editor's `display_frame_added` method can place information in that frame's part info field stating that it needs the selection focus when the window becomes active.

To remove a frame in which a part is displayed, the `remove_embedded_frame` method of the containing part is called to inform it that the part is no longer viewed in that frame. The containing part then calls the frame's `remove` method to delete the reference to the frame. After removing the frame, the containing part calls the part's display_frame_removed method:

The part's `display_frame_removed` method updates the part's internal list of display frames and other related structures to reflect the removal of the frame.

4.5.9.4 Closing and Connecting

For efficient memory use, a containing part is not required to keep frame objects in memory for all of its embedded frames. Instead, it might release frames as they become invisible through scrolling, and reinstantiate them as they become visible.

A containing part that manages frames this way can `close` a frame (remove it from memory but not from storage in the document), and then `connect` it again (read it back from storage and assign it to the part). The containing part must notify the embedded part when it makes these changes. When it closes one of the part's display frames, it calls the part's `display_frame_closed` method:

The part's `display_frame_closed` method updates the part's internal list of display frames and other related structures to reflect the temporary removal of the frame.

When it reinstantiates and reconnects one of the part's display frames, the containing part calls the part's `display_frame_connected` method:

The part's `display_frame_connected` method updates its internal list of display frames and other related structures to reflect the addition of the frame.

4.5.9.5 Synchronization

An embedded part does not directly control the frames in which it is displayed. However, there are times in which an embedded part may need to request that its containing part manipulate the embedded part's display frames for special purposes:

- If a part needs to flow text or other content through a sequence of separate frames in an embedded part (as for page-layout purposes), it can request that the containing part assign the frames to a frame group.
- A part may need one of its display frames to be moved in front of (or behind) another frame belonging to it (or to another part), in order to change the overlap relationship between them.
- Sometimes, views of a part in two or more separate frames must be **synchronized,** meaning that any editing, scrolling, or other changes to one (the **source frame**) must force redrawing of the other. In some cases, such dependencies can be determined internally, but not always. For example, if a containing part creates multiple views of an embedded part, it asks to synchronize those views by calling the part's `attach_source_frame` method:

The part's `attach_source_frame` method takes whatever action is necessary to synchronize the frames. As a minimum, if the two frames have the same presentation, the method should duplicate all embedded frames in one frame into the other (and attach them to their source frames as well).

Frame synchronization is necessary because each display frame of a containing part represents a separate display hierarchy. For invalidating and redrawing, OpenDoc itself maintains no direct connection between embedded frames in those separate hierarchies, even if they are exact duplicates that show the same embedded-part content.

4.5.9.6 Defining General Display Characteristics

OpenDoc parts can display themselves in different ways in different frames, or in different ways in the same frame at different times. The part is in control of its display within the borders of its frames, but there are conventions for other parts to request that the part display itself in a particular way.

There are two kinds of display categories. The **view type** of a frame indicates whether the part within it is shown as one of several kinds of icons (standard icon, small icon, or thumbnail), or whether the part content itself is drawn, within a frame. Some view types are standard values, defined by OpenDoc; any part should be able to draw itself in any of the standard view types. A part can define its own view types also.

The **presentation** of a frame describes, for parts whose view type is framed, how the content is to be represented within the frame. Presentations are part-defined; the part define what types of presentations the part is capable of, and assign their values. Examples of presentations are: text, styled text, bitmap, bar chart, pie chart, tool palette.

View type and presentation are represented as tokenized ISO strings, of type `TypeToken`. If a part creates a presentation type, it is first defined as an ISO string and then the session object's `tokenize` method is used to convert it to a token.

4.5.9.6.1 View Type

A part can get and set the view type of its own frames by calling the frame's `get_view_type` and `set_view_type` methods, respectively.

Note that another part can change a part's frame's view type, or a part can change the view type of another part's frame, by calling the frame's `change_view_type` method. `change_view_type` sets the view type and then notifies the owning part, by calling the part's `view_type_changed` method.

If a part receives this call, it must display itself in the specified frame according to the indicated view type. Parts must support all standard view types.

4.5.9.6.2 Presentation

A part can get and set the presentation of its own frames by calling the frame's `get_presentation` and `set_presentation` methods, respectively.

Note that another part can change a part's frame's presentation, or a part can change the presentation of another part's frame, by calling the frame's `change_presentation` method. `change_presentation` sets the presentation and then notifies the owning part, by calling the part's `presentation_changed` method:

If a part receives this call, and if it supports the indicated presentation, it must display itself in the specified frame accordingly. If it recognizes the indicated presentation but does not support it, or if it does not recognize it, the part must pick a close match or a standard default. It then calls the frame's `set_presentation` method to give it the correct value.

4.5.10 Embedded Frame and Facet Manipulation

The `Draft` object is responsible for creating and initializing frames. A part that needs to create a frame calls its draft's `create_frame` method. The draft, among other tasks, is responsible for calling the frame's `init_frame` method.

Containing parts need to maintain information on the shapes and transforms for all its visible embedded frames and facets. If it makes changes to them, it not only updates its own information, but must in some cases also notify the embedded parts of the changes, so that they can update their own information. Containing parts must also support frame negotiation to permit embedded parts to request additional frames or changes to the sizes of their existing frames.

A containing part must implement an iterator class (type `EmbeddedFramesIterator`) that gives a caller access to each of its embedded frames.

When the user adds a part to a document, that part is embedded within an existing part, the containing part. The part editor for the containing part decides, based on its own content model, what area is to be given to the new part for display: The containing part editor defines a frame shape for the frame and creates an external transform to use for positioning the facets of the embedded frame.

For each of its display frames that is to show the embedded part, the containing part editor creates an embedded frame for the new part. If there is more than one display frame that can show the embedded part, the containing part editor attaches this frame to its source frame. If the embedded part's frame is presently visible, the containing part creates a new facet (or facets) to display the embedded part. The containing part invalidates all facets of this display frame, so the new part will be displayed at the next update.

Either as soon as it creates an embedded frame, or when the active frame changes, the containing part calls the `content_changed` method of each of its display frames, to notify all of its containing parts that its content has changed, so that they can update any link sources that they maintain.

Finally, the containing part calls its draft's `set_changed_from_prev` method. This marks the draft as dirty, giving the part the opportunity to write itself to storage when the draft is closed.

When an embedded part is removed, the containing part editor is responsible for removing the embedded frame and all the facets of that frame, and redisplaying its own content. If the removal is part of a cut operation, the containing part should place the frame shape on the Clipboard along with the cut part.

Containing parts initiate embedding by calling the `create_frame` method of the draft, which returns an initialized frame that has already been assigned to the embedded part (`create_frame` calls the `display_frame_added` method of the embedded part to do that). This ensures that the new frame can be used as soon as it is returned by the draft. If the embedded frame is visible within the containing frame, the containing part must create a facet for it.

A part that supports embedding may need to respond to a request to add an additional embedded frame. An embedded part can call the embedding part's `request_embedded_frame` method in order to get an additional frame for its content. The embedded part passes information on the frame's frame shape, base frame, part, view type, presentation, containing frame, and overlaid status:

The `request_embedded_frame` method passes most of this information along to the draft's `create_frame` method. The base-frame parameter specifies which of the embedded part's existing display frames is to be the base frame for the new frame; the newly frame will be a sibling of the base frame and will be in the same frame group (if any) as the base frame.

The `request_embedded_frame` method might be implemented in such a way that it calls a private embedded-frame-creation method to actually create the frame.

To change a frame's size, the user typically selects the frame and manipulates the frame border's resize handles. The containing part is responsible for drawing the selected frame border, determining what resize handles are appropriate, and interpreting drag actions on them.

If a part is the containing part of a frame that the user resizes, or if a containing part has other reasons to change the size of an embedded frame (for example, to enforce gridding or in response to editing of its own intrinsic content surrounding the frame), it needs to notify the embedded part that its frame has changed. It calls the frame's `change_frame_shape` method and pass it the new shape, and the frame in turn notifies its part by calling its `frame_shape_changed` method. In response, the embedded part may request a different frame size.

Note that a containing part may also have to adjust the layout of its own intrinsic contents, such as wrapped text, as a result of the resizing.

Call the frame's `remove` method to remove a frame embedded in a part.The embedded part is notified of the removal by a call to its `display_frame_removed` method.

A **frame group** is a set of display frames used in sequence. For example, a page-layout part that flows text from one frame into another uses a frame group to accomplish that. Each frame in the frame group has a sequence number; the sequence numbers establish the order of content flow from one frame into the next.

Sequence information is important for a frame group because the embedded part needs to know the order in which to fill the frames, and you probably need to allow the user to set it up**.**

A part creates and maintains the frame groups used by its embedded parts. To create a frame group, call the `set_frame_group` method of each frame in the group, passing it a **group ID**. Also assign each frame a unique **sequence number** within its group, by calling its `set_sequence_number` method. Of course, frames can be added to and removed from frames from the group, and their sequence can be altered, with additional calls to `set_frame_group` and `set_sequence_number`. The embedded part displayed in the frame group can find out the group ID or sequence number by calling `get_frame_group` and `get_sequence_number` on any of its frames.

Sibling frames are frames embedded at the same level within a containing frame. They may be frames in a frame group (described in the previous section), or they may be unrelated frames belong to different embedded parts. Because sibling frames can overlap, it is the responsibility of the containing part to ensure that clipping occurs properly.

A containing part must keep track of its embedded frames and content elements, so that it can reconstruct the overlapping relationships among them.

A part controls the z-ordering among sibling frames embedded in it, and can communicate that information to OpenDoc through the `move_behind` and `move_before` methods of the containing facet of the sibling frames' facets. The facet positioning achieved with `move_behind` and `move_before` is reflected in the order in which facets are encountered when a **facet iterator** (class `FacetIterator`) is used to access each of the sibling facets in turn (to, for example, calculate a resulting clip shape).

If a containing part wants to create multiple similar views of an embedded part, it must ask that part to synchronize those views. Then, if editing takes place in one of the frames, the embedded part will know to invalidate and redraw the equivalent areas in the other frames. The containing part makes that request by calling the embedded part's `attach_source_frame` method. It calls `attach_source_frame` as soon as the frame that needs to be synchronized with the source frame is created.

4.5.10.1 Adding a Facet

OpenDoc needs to know what parts are visible in a window and where, so that it can dispatch events to them properly and make sure they display themselves. But OpenDoc does not need to know the embedding structure of a document: that is, it is not directly concerned with what embedded frames are located where in each of the parts of the document, and what their sizes and shapes are. Because embedded frames are considered to be content elements of their containing part, each containing part encapsulates embedded-frame positions, sizes, and shapes in its own internal data structures. Therefore, containing parts need to give OpenDoc information about embedded frames only when they are visible. They do this is by creating a facet for each location where one of their embedded frames is visible.

An embedded frame may become visible immediately after being created, or when the containing part has scrolled or moved it into view, or when an obscuring piece of content has been removed. However it happens, the containing part must ensure that there is a facet to display the frame:

- If there is no facet already, the containing part makes one by asking the containing facet to create one; it calls the containing facet's `create_embedded_facet` method.
- If the containing frame has more than one facet, the containing part must create an embedded facet in each of those facets. The containing part iterates through the facets of its frame, creating an embedded facet for each.
- The containing part assigns each embedded facet a clip shape, based on the embedded frame's frame shape but also accounting for any obscuring content in the containing part.
- The containing part assigns each embedded facet an external transform, taken from the containing part's internal data on the location of the frame.

After creating the facet, the containing facet calls the embedded part's `facet_added` method to notify it that it has a new facet.

4.5.10.2 Removing a Facet

An embedded frame may become invisible when the containing part has deleted it, scrolled or moved it out of view, or placed an obscuring piece of content over it. However it happens, the containing part can then delete the facet, because it is no longer needed.

A part deletes the facet of an embedded frame by calling the containing facet's `remove_facet` method. It then calls the embedded part's `facet_removed` method to notify it that one of its facets has been removed. (If the frame that is no longer visible has more than one facet, it needs to iterate through each one, removing it in turn if it is not visible.)

The facet does not actually have to be deleted from memory the moment it is no longer needed; it can instead be marked privately as unused, and not actually be removed until the `purge` method is called.

4.5.11 Focus Manipulation

A part editor uses the `Arbitrator` object methods to acquire a focus. OpenDoc calls three `Part` object methods when it wants a part to relinquish a focus. The `begin_relinquish_focus` method is called to start the process. The part returns a value that indicates whether or not it is able to relinquish the specified focus. The `abort_relinquish_focus` method is called to inform a part that OpenDoc has decided that it does not want a part to relinquish the focus indicated by the `begin_relinquish_focus` call. The can happen because the focus was part of a focus set, and the holder of one of the resources in the focus set indicated that it could not relinquish the focus. OpenDoc calls the `commit_relinquish_focus` method to inform a part that it must relinquish a focus; this happens if every holder of a focus in a set has indicated that it can relinquish the focus.

Two other methods, `focus_acquired` and `focus_lost` are called by OpenDoc to inform a part that one of its display frames has either acquired or lost a particular focus.

4.5.12 Menus

OpenDoc manages a focus (see "Focus" on page 35) for the menu bar. OpenDoc calls the part's `adjust_menus` method when the part has the menu focus and there is a mouse down event in the menu bar. The method can call the `MenuBar` object methods to modify the menu bar.

4.5.13 Event Handling

Events in OpenDoc include mouse clicks and keystrokes, menu commands, activation and deactivation of windows, and other events available on only some platforms.OpenDoc passes user event information to a part by calling its `handle_event` method.

The document shell receives equivalents to OpenDoc user events through whatever mechanism the underlying platform provides and converts them into the form of OpenDoc user-events and passes them to the `Dispatcher`. The `Dispatcher` handles the events it recognizes, using a `DispatchModule` to pass events to individual parts. The `Dispatcher` returns all other events to the document shell to handle.

A part becomes the target for a specific type of user event (other than mouse events) by obtaining the *focus* for that event. The OpenDoc arbitrator decides which part is to receive an event by consulting a `FocusModule`, which tracks, for example, which part is currently active and therefore should receive keyboard events.

Geometry-based events such as mouse clicks are generally dispatched to the parts within which they occur, regardless of which part currently has the selection focus—that is, regardless of which part is currently active. This permits inside-out activation to occur.

A part receives information about its events in a platform-specific **event structure.**

The inverse of the drawing coordinate transformation is necessary to assign content locations to mouse events.

If a mouse click occurs in a facet of a part's frame, OpenDoc applies the inverse of all external transforms from the root facet to the facet, passing the mouse event in the part's frame coordinates. It is then the part's responsibility to apply the inverse transform to the event location, to convert it to the content coordinates of the part.

If the entire content of a part's frame is scrolled, the application of the frame's internal transform yields the correct location for everything within the frame.

Part editors need to respond to events that select or change the position or visibility of its intrinsic content, by highlighting or preparing to move or redraw the content.

Part editors are responsible for knowing what embedded frames look like and when they become visible or invisible. When update events or events related to scrolling or editing cause an embedded frame to become visible, a part is

responsible for creating facets for those frames. If events cause an embedded facet to move, a part must modify the facet's external transform to reflect the move. If the embedded facet moves so as to become no longer visible, a part is responsible for deleting the embedded facet from its containing facet (or marking it as purgable).

Note that user events are platform-specific. The following subsections give examples for classes of events that are expected to be common across platforms.

4.5.13.1 Mouse Events

When the user presses a mouse button while the mouse pointer is within a facet of a part, the dispatcher calls the part's `handle_event` method, passing it a mouse down event. When the user releases a mouse button while the pointer is within a facet, the dispatcher calls the part's `handle_event` method, passing it a mouse up event.

The event-dispatching code does not itself activate and deactivate the relevant parts; it is up to the editor of the part receiving the mouse event to decide whether to activate itself or not.

When the user presses or releases the mouse in the content area of an OpenDoc window, the dispatcher finds the correct facet for handling the mouse event by traversing the hierarchy of facets depth-first (trying the most deeply embedded facets first) and front-to-back (trying the frontmost of sibling facets first). The dispatcher sends the event to the editor of the most deepest-embedded and frontmost frame it finds whose active shape contains the mouse position. In this way the smallest enclosing frame surrounding the mouse location gets the mouse event, preserving the OpenDoc inside-out activation model. None of the part editors of any containing parts in that frame's embedding hierarchy are involved in handling the event.

4.5.13.1.1 Mouse Events in Embedded Frames

If a part contains embedded frames, the dispatcher can also send it special mouse events that occur within and on the borders of the embedded frames' facets.

4.5.13.1.2 Keyboard Events

When the user presses a key on the keyboard and a part has the keyboard focus, the dispatcher calls the part's `handle_event` method, passing it a key down event. When the user releases the key, the dispatcher again calls the part's `handle_event` method, this time passing it a key up event.

Exceptions to this include keyboard events that are command equivalents, which go to a part as menu events, and keyboard events involving special navigation keys such as Page Up, Page Down, Home and End keys, which go to the frame—if any—that has the scrolling focus.

4.5.13.1.3 Menu Events

If the user presses a mouse button when the mouse pointer is within the menu bar, or if the user enters a keyboard equivalent to that action, OpenDoc converts the mouse-down or keyboard event into a menu event, and calls the `handle_event` method of the part with the menu focus.

On platforms whose operating systems support unique menu-command IDs, the part's `handle_event` method is passed the command ID for the menu event. On other platforms, the `handle_event` method can obtain a command ID for the menu event by calling the `get_command` method of the menu bar object.

4.5.13.1.4 Window Events

If the user presses a mouse button when the mouse pointer is within the title bar of a window (such as when clicking in a close box), or if the user enters a keyboard equivalent to that action, OpenDoc converts the mouse-down or keyboard event into a window event and calls the `handle_event` method of the window's root part.

Normally, the root part does not handle the event, and the dispatcher returns it to the document shell, which performs the intended action (such as closing the window). However, a root part can handle the event if it wishes. For example, it might hide the window rather than closing it.

4.5.13.1.5 Activation Events

On platforms that support the notion of activation and deactivation of windows, OpenDoc sends activate events and deactivate events to each facet in the window when the window changes state.

When an active facet of a frame of a part becomes inactive through window deactivation, the part's `handle_event` method can—upon receiving the deactivation event—store a flag in the facet's part info field to note that the facet was active before window deactivation. Then, when a facet of a frame of the part receives an activation event because of window activation, the part's `handle_event` method can examine the part info field to decide whether or not the part should activate itself.

4.5.13.2 Propagating Events

A containing part can set the `does_propagate_events` flag of an embedded frame that it creates. If it does, the containing part receives events not handled by the embedded frame. If a part contains embedded frames with that flag set, the `handle_event` method receives the events originally sent to them.

4.5.14 Undo

Undo (and its reverse, Redo) is a feature that allows the recovery from user errors. OpenDoc, in cooperation with part editors, supports multiple levels of Undo and Redo, including inter-document Undo and Redo. OpenDoc does not limit the number of times that Undo can be invoked in succession.

The OpenDoc Undo feature does not offer infinite recoverability, of course. Some user actions clear out the Undo **action history**, the cumulative set of reversible actions available at any one time. A typical example is saving a window; Undo cannot unsave it, and no actions executed prior to its saving can be undone. Part developers decide which of their parts actions are undoable, which ones are ignored for Undo purposes, and which ones reset the Undo history. In general, actions that do not change the content of a part (such as scrolling, making selections, and opening and closing windows) do not need to be undoable.

When the user selects Redo, it reverses the effects of the last Undo. Redo is available only if the last undoable user action was invoking Undo or Redo. Like Undo, Redo can be invoked several times in succession, limited only by the number of times Undo has been invoked in succession. As soon as an undoable action (such as an edit) is performed, it clears the Redo history and Redo is no longer available until another Undo is invoked.

To implement this multilevel Undo that spans different parts and different documents, OpenDoc maintains a centralized repository of undoable actions, stored by individual part editors as they perform such actions. Undo history is stored in the `Undo` object, an instantiation of the class `Undo`, created by OpenDoc and accessed through the session object's `get_undo` method. A part editor needs access to the `Undo` object in order to store undoable actions in it or retrieve them from it.

Parts should support multiple levels of Undo. If a part supports only a single level of Undo, all undoable commands that it executes result in clearing the Undo history for all of OpenDoc. Other parts that support multilevel Undo would lose their Undo history when they interact with a part that does not, but not when interacting with each other. To avoid this inconsistency, multiple levels of Undo should be supported if Undo is supported at all.

To implement support for Undo, a part editor must save information in the Undo object that allows it to recover previous states of the part, and the part editor must implement five methods.

The methods `undo_action` and `redo_action` of the part are called by the `Undo` object when the user selects the Undo and Redo items from the Edit menu, respectively.

The methods `write_action_state`, `read_action_state`, and `dispose_action_state` of a part are called by the `Undo` object when it needs it to persistently store or retrieve undo-related information. They allow the Undo action history to be saved in low-memory situations, and retrieved later.

If a part performs an undoable action, it should call the `Undo` object's `add_action_to_history` method. It passes an item of **action data** to the method; the action data contains enough information to allow the part to revert to the state it occupied just prior to the undoable action. The action data can be in any format; it is added to the action history in the `Undo` object, and is passed back to the part if the user asks the part to perform an Undo.

Two user-visible strings are also passed to the `add_action_to_history` method. The strings have text to appear on the Edit menu, such as "Undo Cut" and Redo Cut". A part must also specify the data's action type. A part decides what constitutes a single action.

Some transactions, such as drag-and-drop, have two stages—a beginning and an end. To add such a transaction to the undo history, a part must define which stage it is adding, by using the **action types** `begin_action` and

end_action. (For undoable actions that do not have separate stages, specify an action type of single_action when calling add_action_to_history.)

In the case of drag-and-drop, the source part specifies an action type of begin_action when calling add_action_to_history, and the destination part specifies end_action when calling add_action_to_history.

Note that parts can add single-stage actions to the undo history during the time that a two-stage undoable action is in progress—that is, between the times add_action_to_history is called with begin_action and with end_action, respectively. These actions become part of the overall undoable action as well.

It is recommended that the undoable actions performed while in a modal state not affect the action history previous to that modal state. For example, if a part displays a modal dialog that does not perform any actions that would clear the Undo history of the document the part belongs to, the user should be able—once the modal dialog goes away—to undo actions performed before the modal dialog box was displayed (even though the user could not, at this point, undo actions taken while the modal dialog was open).

In order to implement this behavior, put a mark into the action history that signifies the beginning of a new **action subhistory.** Do that by calling the mark_action_history method of the Undo object. To clear the actions within a subhistory, specify respect_marks when calling the clear_action_history method; to clear the whole action history instead, specify dont_respect_marks.

For every time a mark is put into the action history, make sure there is an equivalent call to clear_action_history to clear that subhistory.

When the user chooses to undo an action added to the action history by a part, OpenDoc calls the part's undo_action method, passing it the action data that was passed in earlier. The part then restores itself to the pre-action state. (When a two-stage transaction involving a part is undone, OpenDoc calls both the part and the other part or parts involved, so that the entire compound action—including any single-stage actions that occurred between the begin and end actions—is reversed.)

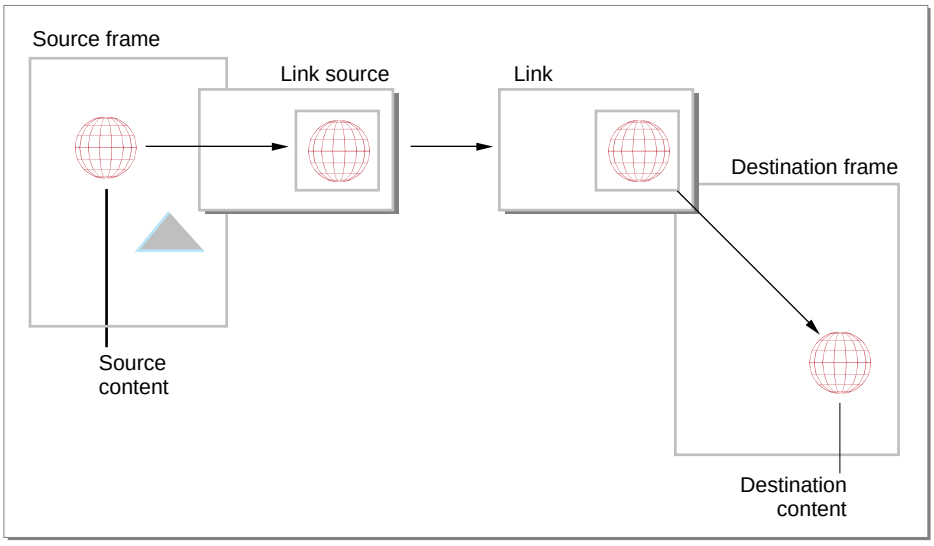
When the user chooses to redo an action of a part, OpenDoc calls the part's redo_action method, passing it the same action data that had been passed in earlier, when the original action was performed. In this case, a part's task is to reachieve the state represented by the action data, not reverse it. (When a two-stage transaction involving a part is redone, OpenDoc calls both part parts involved, so that the entire compound action—including any single-stage actions that were undone between the begin and end actions—is restored.)

As soon as a part performs an action that cannot be undone it clears the Undo history by calling the Undo object's clear_action_history method.

4.5.15 Linking

Linking is a mechanism for placing information into a part by reference. Linking requires the cooperation of one or two parts with several linking-related objects. Figure 45 illustrates schematically the objects and data involved. The figure applies most closely to a link between two separate parts in the same document, although links can occur within a single part and also between parts in separate documents.

Figure 45 Objects and data involved in linking



The part that contains the source of the information to be linked is called the **source part**. The content that is to be copied and sent to another part is called the **source content** or **source.** The source part creates a `LinkSource` object that contains a copy of the source data and is stored in the same document as the source part.

The part that contains the destination of the information to be linked is called the **destination part.** The content that is actually copied into the destination part is called the **destination content** or **destination.** The destination part's draft object creates a `Link` object that is stored in the same document as the destination part.

The `LinkSource` object contains references to all link objects for which it is the source. (Within a given document, there is only one link object associated with each link source, even if the source is linked to multiple destinations in the document.) When a change occurs in the source part and the destination part needs to be updated, the source part copies the source content into the `LinkSource` object, and the destination part copies the content from the `Link` object into the destination itself.

Whether it contains the source or destination of a link, a part is responsible for defining the exact content that constitutes the linked data, and for knowing what visual area it covers in the part's display. Linked data, whether source or destination, is part of the part's own content; it is stored and manipulated it according to its own content model.

A link is formed when the user designates a part as the destination of a link. A link can only be made to data that has been advertised as being available for a link. A part editor advertises that it has content available for linking by creating a `LinkSpec` object, which a it makes by calling `Draft::create_link_spec`. The part that wishes to receive a link creates a `Link` object by calling `Draft::acquire_link`, passing the `LinkSpec` for the data that it desires. OpenDoc then calls the source part editor's create_link method which then makes a `LinkSource` object by calling `Draft::create_link_source`. The source data is made available to the destination using storage units (see 4.4.1, "Storage") attached to the link objects.

Source parts call `LinkSource::content_updated` when they have modified data that is the source of a link. This results in OpenDoc calling the destination `Part::link_updated` method if automatic updating has been enabled by calling `LinkSource::register_dependent`, the implementation of which can handle the new data.

OpenDoc call the `embedded_frame_updated` method of containing parts to when an embedded part has changed. Containing parts update any relevant link source data in their implementation of this method.

OpenDoc calls the `edit_in_link_attempted` method of a part to find out if a particular frame that belongs to a part contains the destination of a link.

Frames have a link status which indicate whether the frame contains a link source, a link destination, or no link. When the link status of a part's frame changes, OpenDoc informs the part by calling its `link_status_changed` method.

The source of a link may not be visible on the screen. For example, a chart in a document that is a link source may be scrolled out of view. When OpenDoc calls a part's `reveal_link` method the part must make the link source content visible on the screen. A part editor may choose to highlight this content so that it can be recognized by the user.

4.5.16 Drag and Drop

The OpenDoc drag-and-drop facility allows users to apply direct manipulation to move or copy data. Users can visually move images that represent information on the screen causing the content to be moved or copied.

4.5.16.1 User Interaction

The user typically initiates a drag by positioning the mouse pointer over some selected content, and pressing and holding down the mouse button. The part that owns the content makes a copy of the dragged item, places its image at the mouse pointer location, and selects it. A modifier key may be pressed before the mouse button to force a drag to be a copy (called a **drag-copy**) or a move (called a **drag-move**).

As the user moves the mouse pointer, an outline of the selected item is dragged to the new location. When the user releases the mouse button, the item is placed (dropped) at the mouse pointer location. The part beneath the mouse pointer receives a notification that something has been dropped on it. At this point the operation performed by the destination part is essentially identical to pasting from the Clipboard, including considerations of embedding vs. incorporating and handling of linked data. The source part may have to delete the items if it was a move operation, or do nothing if it was a copy.

Drag-and-drop allows parts to restrict the dropped content they will accept to specific part kinds. To indicate that it can accept a drop, a part provides appropriate feedback to the user once the mouse pointer enters its facet.

For frames and icons that are being dragged, OpenDoc provides specific behavior and appearance for the item being dragged. For intrinsic content that is being dragged, the part editor is responsible for defining behavior and providing user feedback.

A frame can be moved by dragging its border. The user moves the mouse pointer to the border of an active or selected frame, and presses the mouse to start dragging the frame. When the user releases the mouse button, the frame and its part are dropped at the new location. After a frame is dropped, it becomes selected.

The destination may adjust the drop location of a frame based on constraints such as gridding (in a graphics part) or text position (in a text part).

Ordinarily, a frame can be dragged only by its border, but if a frame is bundled it can be dragged by pressing the mouse button down anywhere in its interior, and it does not have to be activated or selected first.

4.5.16.2 Move vs. Copy

OpenDoc follows these conventions for drag-and-drop. When a user drags an item within a document, the item is moved, not copied. When the user drags an item across a document boundary, the item is copied, not moved. Note that a window boundary is not always a document boundary; dragging an item to or from a part window, but still within the same document, results in a move.

When using drag-and-drop to move a frame, if the destination is in the same document, the moved frame displays the original part. However, if the destination is in a different document, the source frame is deleted and a new part (and new frame displaying the contents) is created at the destination. In contrast, Clipboard pasting always creates a new part and frame.

4.5.16.3 Dragging Stationery

Stationery parts may be copied exactly like any other part, but a move into a part where the preferred view type for embedded parts is a frame results in a copy being “torn off” the stationery pad and opened into a frame. The stationery part remains in its original location, unchanged.

4.5.16.4 Initiating a Drag

In response to a mouse-down event, such as on the border of one of its embedded frames or on a selected set of content elements, the source part has the choice of initiating a drag. To do so, the source part follows these steps:

- It calls the `get_drag_and_drop` method of the `Session` object to get access to the `DragAndDrop` object. It then calls the `clear` method and the `get_content_storage_unit` method of the drag-and-drop object, to get an empty storage unit into which the dragged data is to be copied.
- It writes the dragged data into the drag-and-drop storage unit, just as when writing data to the Clipboard. Like the Clipboard, the drag-and-drop object can hold intrinsic data only, a single embedded frame, or a combination of intrinsic data plus embedded frames (see Figure 35 on page 47):
  - If the data is intrinsic content, the source part follows a procedure similar to that listed under "Copying Intrinsic Content" on page 48.
  - If the data is a single embedded part, the source part follows a procedure similar to that listed under "Copying a Single Embedded Part" on page 48.
  - If the data consists of intrinsic content plus one or more embedded parts, the source part follows a procedure similar to that listed under "Copying Content That Includes an Embedded Frame" on page 49.
  - If the data contains linked data (link destinations, link sources, or both), the source part follows a procedure similar to that listed under "Copying Linked Content" on page 49.
- It initiates the drag by calling the drag-and-drop object's `start_drag` method.

The source part is responsible for providing the user an image, such as an outline, for dragging feedback.

4.5.16.5 Operations While Dragging

While the user holds the mouse button down, the drag is in progress. Potential destination parts need to perform certain actions during dragging, when the mouse pointer is within their facets.

4.5.16.5.1 On Entering a Part's Facet

Any facet the mouse pointer passes over during a drag represents a potential destination for a drop. OpenDoc calls a part's `drag_enter` method when the mouse pointer enters one of its facets during a drag. The potential destination part should examine the part kinds of the dragged data (using a drag-item iterator passed to it), and determine whether or not it can accept the dragged data. If the potential destination part can accept a drop, it should provide the appropriate feedback to the user, such as adorning its frame border or changing the cursor appearance. If the potential destination part cannot handle the part kinds in the dragged data, it should take no action.

4.5.16.5.2 While Within a Part's Facet

OpenDoc calls the potential destination part's `drag_within` method continuously while the mouse pointer is still within the facet. This allows the part to do any desired processing inside of its display. For example, if a part allows objects to be dropped only in individual hot spots, it may change its feedback based on mouse-pointer location.

- Calls to a part's `drag_within` method also give it a chance to examine the state of the machine. For example, the part may want to find out whether or not a modifier key is down while a drag is in progress.

4.5.16.5.3 On Leaving a Part's Facet

OpenDoc calls the potential destination part's `drag_leave` method when the mouse pointer leaves a droppable facet. This allows the part to remove the adornment on its frame or change the cursor back to its original form.

4.5.16.6 Dropping

If the mouse button is released while the pointer is within a facet, OpenDoc calls the `drop` method of the facet's part. The potential destination part then decides whether it can receive the dragged object, using the drag-item iterator passed to it to examine the dragged data's part kinds. If the destination part accepts the data, the process is similar to incorporating or embedding data from the Clipboard, as described in the section "Pasting Data from the Clipboard" on page 50. If the destination part needs to put up a Paste As dialog box, it can follow the procedure outlined in the section "Paste As" on page 100.

The destination part should return an appropriate result (of type `DropResult`) from its `drop` method. OpenDoc in turn tells the source part whether the drop was accepted. The source part can then, for example, delete the data if the drag operation was a move rather than a copy.

4.5.16.7 Drag-Item Iterators and Non-OpenDoc Data

OpenDoc allows for platform specific mechanisms to permit interchange with non OpenDoc data.

On the Macintosh, for example, when a part's `drop` method is called, it is passed a drag-item iterator (class `DragItemIterator`), so that it can access all drag items in the drag-and-drop object. If it is OpenDoc data that has been dragged, there is only one drag item in the object, but if the data comes from outside of OpenDoc, there may be more than one item. It is the part's responsibility as destination to iterate through all the drag items to find out whether it can accept the drop.

A part can examine the contents property of the storage unit of each dragged item to determine the kind of data it consists of.

4.5.16.8 Promises

Using a promise in drag-and-drop (or in Clipboard transfer) requires the source part to first put out the promise, and then fulfill it when the data is needed.

4.5.16.8.1 Putting Out a Promise

A source part can put out a promise in response to a mouse down event that initiates a drag, or when copying data to the Clipboard. Follow these steps:

1. Gain access to the storage unit into which data for the drag or copy operation is to be stored. See "Initiating a Drag" on page 88 for drag-and-drop; see, for example, "Copying Intrinsic Content" on page 48 for Clipboard transfer. Focus the storage unit to the contents property (`prop_contents`).
2. Prepare your promise. It can have any content and formats as long as it can be read later and the exact data that is to be transferred can be reconstructed.
3. Write the promise into a single value of the storage unit, using its `set_promise_value` method.
4. Initiate the drag, or complete the transfer of information to the Clipboard. Clone any frames or creating a link spec or reading the change ID is deferred until the promise is fulfilled.

4.5.16.8.2 Getting Data From a Value with a Promise

When the `drop` method of a destination part retrieves the data from a drop or from the Clipboard (using the `get_value` or `get_size` methods of the dragged data's or Clipboard's storage unit), the source part is called to fulfill the promise. The destination part does not even know that the fulfillment of a promise is taking place; it uses the same code whether the value contains a promise or not.

4.5.16.8.3 Fulfilling a Promise

OpenDoc calls a source part's `fulfill_promise` method when a promise must be fulfilled, passing it a storage-unit view object that contains the promise. An implementation of the method might do something like the following:

1. Using the private information that was earlier written into the promise, retrieve the actual data that the promise represents. Clone frames and create a link spec as usual at this time.
2. Write the actual data back into the storage-unit view object.

When the `fulfill_promise` method completes, the destination part receives the data in the storage-unit view.

4.5.17 Container Properties

The properties associated with the containing part of an embedded part can change. The embedded part's `containing_part_properties_updated` method is called by OpenDoc to notify the embedded part that its containing part's properties have changed.

A containing part uses the `acquire_containing_part_properties` method to get the properties associated with one of its embedded parts.

4.6 User Events and User Interface

An OpenDoc part editor is required to respond to a specific set of actions or messages from OpenDoc that, although generated and passed in different ways on different platforms, constitute the majority of user interactions with the parts. In OpenDoc, these messages are called **user events.**

It is by handling user events that a part editor interacts with the user, and cooperates with the user in constructing compound documents.

4.6.1 Activation and Focus Transfer

This section discusses event-handling and related tasks involved with activating and deactivating parts, and transferring focus from one part to another.

4.6.1.1 Focus Types and Focus Modules

A part makes itself the recipient of a certain type of user event (other than mouse events) by obtaining the focus for that type of event. A **focus** is a designation of general event type, used to claim ownership; for example, the frame that owns the keyboard focus receives all keyboard events, until it passes ownership of the keyboard focus to another frame.

Focus types are defined as ISO strings, and the standard set defined by OpenDoc includes those in Table 2.

Table 2 Focus types

Constant	ISO string	Description
`key_focus`	“Key”	Keyboard events are sent to the frame with this focus.
`menu_focus`	“Menu”	Menu events are sent to the frame with this focus.
`selection_focus`	“Selection”	Shift-click and Command-click mouse events are sent to the frame with this focus. OpenDoc draws the active frame border around all facets of this frame.
`modal_focus`	“Modal”	The frame that owns this focus is notifying other frames that it is the only current modal frame.
`scrolling_focus`	“Scrolling”	Scrolling-specific keyboard events (such as Page Up and Page Down) are sent to the frame with this focus.
`clipboard_focus`	“Clipboard”	The frame that owns this focus has access to the clipboard.

Foci are owned by frames. Note that, in general, mouse events anywhere within the content area of an OpenDoc window always go to the most deeply embedded frame that encloses the click point, and thus there is no “mouse focus.” However, Shift-clicks and Command- clicks, regardless of their location, are sent to the frame with the selection focus, to allow for extending selections.

OpenDoc does not require that the same frame own the selection focus, keystroke focus, and menu focus, although this is most often the case. Nor does OpenDoc require that the selection focus be in an active window, although this is usually the case on some platforms.

A part needs to convert focus names into tokens before using them in any method calls. Call the `tokenize` method of the session object to convert ISO strings to tokens.

Foci may be manipulated singly, or in groups called focus sets. A **focus set** is an OpenDoc object (of class `Focus-Set`) that lists a group of foci that a part editor wants to obtain or release as a group.

A part editor can define additional focus types as needed. It can define other kinds of focus, perhaps to handle other kinds of user events (such as input from new kinds of devices). To create a new kind of focus, create a new kind of **focus module,** the OpenDoc object that the arbitrator uses to determine focus ownership.

Foci may be exclusive or non-exclusive foci. All of the standard foci defined by OpenDoc are **exclusive,** meaning that only one frame at a time can own a focus. But if a new kind of focus is created, a part can make it **nonexclusive,** meaning that several frames could share ownership of it.

4.6.1.2 Part Activation

Part activation is the process of making a part and frame ready for editing. Activation typically occurs when the user clicks the mouse button when the pointer is within a frame that is not currently active, but it can also happen when the window displaying a part is opened, or when the window is activated, or when a part is pasted from the Clipboard or dropped in a drag-and-drop operation.

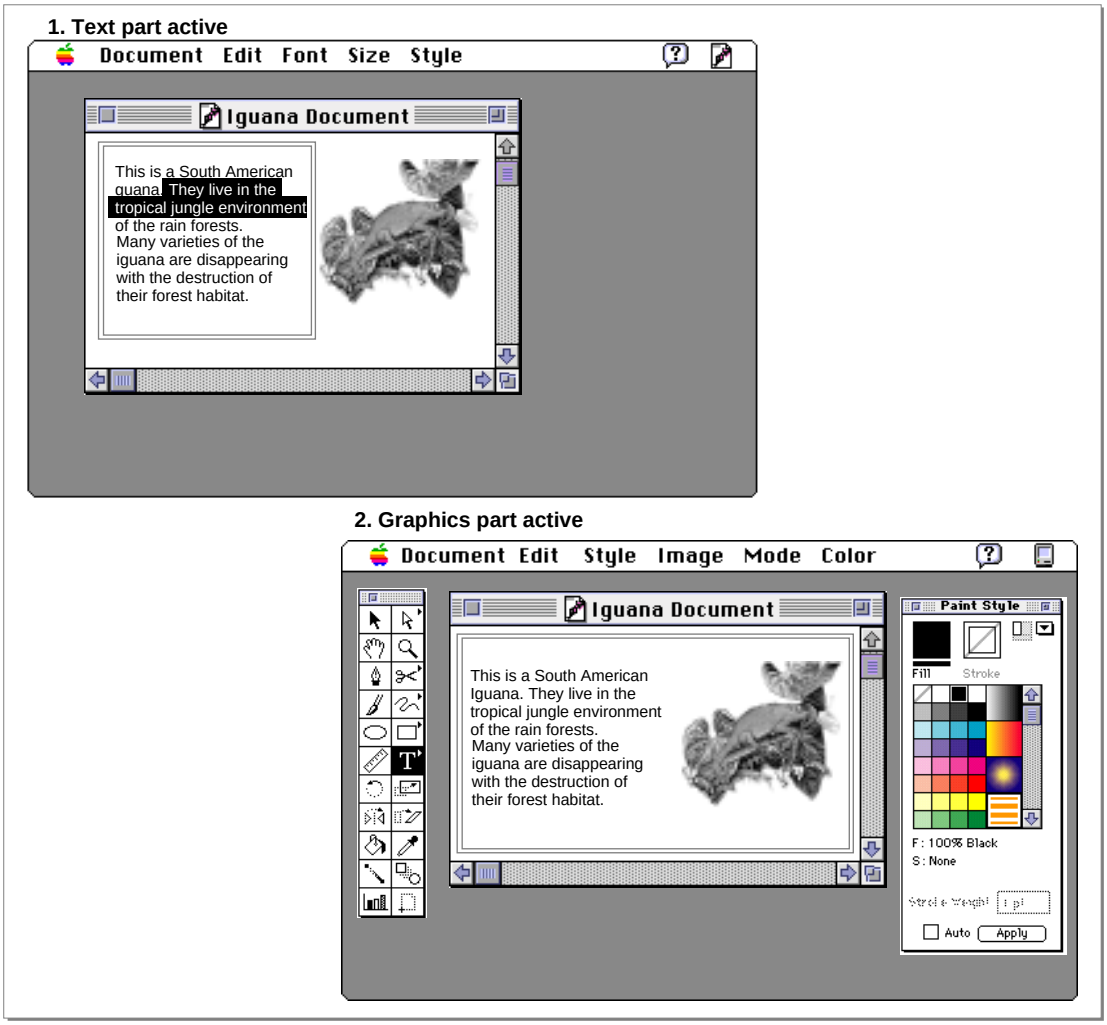
OpenDoc has a flexible and platform-neutral model of part activation, in which part editors activate and deactivate themselves, rather than being activated and deactivated by OpenDoc. To obtain event foci, part editors request them by name from the arbitrator.A part an gets access to the arbitrator by calling the `get_arbitrator` method of the session object.

A part's frame activates itself when it receives a mouse event within its content area, or when its window first opens (or when it becomes active) and the part has stored information saying that the frame should be active on opening or upon window activation.

The part that activates itself is then responsible for displaying its contents appropriately (such as by re-highlighting a selection) and providing any menus, controls, palettes, floating windows, or other auxiliary items that are part of its active state. The part must also obtain ownership of any foci that it needs. OpenDoc draws the active frame border around the active part.

In Figure 46, for example, the text part is active in the first picture. OpenDoc draws the active frame border around its frame, it has a highlighted selection, and it displays Font, Size, and Style menus. In the second picture, the text part has become inactive and the graphics part has become active. The text part has removed its menus and its highlighting. The graphics part, conversely, has put up its own menus and has displayed two palettes (in separate windows). OpenDoc now draws the active border around the graphics part's frame.

Figure 46 Inactive and active states of a graphics part



When one part is activated, another part is usually deactivated. That part knows that it is being deactivated when it receives a request to relinquish the selection focus, plus possibly other foci. The deactivating part is responsible for unhighlighting its selections and removing any menus and other items that were part of its active state. OpenDoc removes the active frame border from around the (now inactive) part, to draw it around the part that currently has the selection focus.

In most cases, when a frame is activated the part editor for that frame requests the selection focus, keystroke focus and menu focus. A frame with scroll bars might also request the scrolling focus. (If the click is within the area of a selection, the activation may not be done on mouse-down, because the user might begin a drag-and-drop operation.)

A simple part, such as a small text-entry field in a dialog box, for example, might request only the selection focus and keystroke focus on receiving a mouse-down within its frame area. An even simpler part, such as a button, for example, might not even request the selection focus. It might simply track the mouse until the button is released, and then run a script, never having changed the menu bar, put up palettes or rulers, or become active.

4.6.1.3 Transferring the Focus

In essence, a part activates itself by requesting ownership of a set of foci, and deactivates by relinquishing ownership of those foci. This section discusses how to request or relinquish foci, as frames are activated or deactivated.

4.6.1.3.1 Requesting Foci

A part can request, for one of its frames, ownership of a single focus or a set of foci. It requests a focus by calling the arbitrator's `request_focus` method; it requests a focus set by calling the arbitrator's `request_focus_set` method. If the request succeeds, the part's frame obtains the focus or focus set.

The arbitrator's `request_focus` and `request_focus_set` methods perform a two-stage action in transferring a focus or focus set:

1. The arbitrator first asks the current owning frame of each focus if it is willing to relinquish the focus, by calling the `begin_relinquish_focus` method of the frame's part.
2. If any owner of the focus is unwilling to relinquish it, the arbitrator aborts the request by calling each part's `abort_relinquish_focus` method. `request_focus` or `request_focus_set` then returns `FALSE`.
3. If all focus owners are willing to relinquish, the arbitrator calls each part's `commit_relinquish_focus` method. `request_focus` or `request_focus_set` then returns `TRUE`.

4.6.1.3.2 Relinquishing Foci

A part does not usually relinquish a focus or set of foci until another part asks for them. Nevertheless, most parts willingly relinquish the common foci when asked.

Relinquishing foci can be a two-step process, because multiple foci of a focus set must all be relinquished at once. A part editor accomplishes this in its `begin_relinquish_focus`, `commit_relinquish_focus`, and `abort_relinquish_focus` methods:

- In the `begin_relinquish_focus` method, the part needs do nothing other than return `TRUE` or `FALSE`, based on the type of focus and the identities of the frames (current and proposed focus owners) passed to it. In most cases it can simply return `TRUE`, unless the part is displaying a dialog box and another part is requesting the modal focus; in that case, because the part does not want to yield the modal focus until its dialog box window closes, it return `FALSE`. See "Acquiring and Relinquishing the Modal Focus" on page 95 for more information.
- In a part's `commit_relinquish_focus` method, it has actually relinquished the focus type responded to in `begin_relinquish_focus`, and takes appropriate action, such as removing menus or palettes, disabling menu items, removing highlighting, and performing whatever other tasks are part of losing that type of focus. Remember that the focus may possibly be moving from one frame to another of the same part, so the exact actions can vary.
- If, after a part responds with `TRUE` to `begin_relinquish_focus`, the focus is actually not transferred from your frame, OpenDoc calls the part's `abort_relinquish_focus` method. If the part has done anything more than return the Boolean result in `begin_relinquish_focus`, it can undo those effects in the `abort_relinquish_focus` method.
- If a part is one of several focus owners called to relinquish a focus set, and if it returns `FALSE` to `begin_relinquish_focus`, the `commit_relinquish_focus` method will not be called (because it chose not to give up the focus), but the `abort_relinquish_focus` method will still be called (because all owners of a focus set are notified if any one refuses to relinquish the focus).

4.6.1.3.2.1 Relinquishing Foci When Closing Frames

In s part's `close_display_frame` and `remove_display_frame` methods, it should include a call the arbitrator's `relinquish_focus_set` method, to relinquish any foci owned by the frame it is closing.

4.6.1.3.3 The focus_acquired and focus_lost Methods

Part editors are expected to implement versions of the methods `focus_acquired` and `focus_lost`. These methods perform any actions your part editor deems appropriate to having just acquired, or having just lost, a specified focus.

During the normal process of requesting and relinquishing foci, a part editor's `focus_acquired` and `focus_lost` methods are not called. In some situations, however, a part editor's `focus_acquired` method might be called directly by the arbitrator; see the next section, "Transferring Focus Without Negotiation." Currently, OpenDoc does not call `focus_lost`.

4.6.1.3.4 Transferring Focus Without Negotiation

The arbitrator's methods `transfer_focus` and `transfer_focus_set` allow a part to transfer focus ownership without negotiation. Parts can use these calls to transfer focus among parts and frames they have direct control over. For example, in a modal dialog box consisting of several parts, these methods can be used to transfer focus from the outer part (the dialog box) directly to an inner part (such as a button) and back. In this case, the arbitrator simply calls the `focus_acquired` method of the new owner.

4.6.1.4 Recording Focus Transfers

OpenDoc does not save or restore focus assignments. During deactivation of windows and frames, and during closing of windows, a part can record the state of focus ownership in order to restore it at a later activation or reopening.

4.6.1.4.1 Focus Transfer on Frame Activation

When the user clicks to activate a previously inactive frame in a window, the part editors involved in the activation simply need to record that a frame has gained or lost the selection focus:

- If a part's frame is being activated, the part editor might record, in the frame's part info field, the fact that the frame has the selection focus. The part editor can perform this action in its `focus_acquired` method and/or after its call to the arbitrator's `request_focus_set` method succeeds.
- If a part's frame is being deactivated, the part editor might record, in the frame's part info field, the fact that the frame does not have the selection focus. The part editor can perform this action in its `focus_lost` method and/or its `commit_relinquish_focus` method.

4.6.1.4.2 Focus Transfer on Window Activation

On some platforms, if the user clicks in the title bar of an inactive window, that window is activated and brought to the front, and the previously active window is deactivated. Each part in the first window receives a deactivate event for each facet of each frame it has in that window. A part's method for handling a deactivate event should find out if the frame has the selection focus, and if so, store—possibly in that frame's part info field—the fact that that frame needs the selection focus the next time the window is activated.

Likewise, a part's method for handling an activate event could examine the frame's part info field, and if it needs the selection focus, request the focus using the arbitrator's `request_focus` or `request_focus_set` methods.

If no embedded frame has the focus on deactivation, the root frame may wish to acquire the focus on reactivation. Activate events are sent bottom-up to the facets in a window, meaning that the root facet receives the event last, so that it can find out at that point whether any other facet has acquired the focus.

4.6.1.4.3 Saving and Restoring the Selection Focus

When the state of a window is saved in a document and the document is subsequently reopened, the window (plus the selection, if it has been saved persistently) is recreated by the window state object. If a part wants to save the selection focus state of its window, and it has stored that information in the frame's part info field with its `add_display_frame` method, it can write it to storage in the part editor's `write_part_info` method when the window is closed, and restore it in the part editor's `read_part_info` method when the window is opened. The part's `write_part_info` and `read_part_info` methods are called by its display frame.

4.6.2 Windows and Dialog Boxes

This section discusses how events are handled in windows, modal dialog boxes, movable modal dialog boxes, and modeless dialog boxes.

4.6.2.1 Windows

In order to receive events in its windows, a part must create an `Window` object for each platform-specific windows it uses, including dialog boxes.

4.6.2.1.1 Activating and Deactivating Windows

Some platforms have a notion of active and inactive windows. An active window is brought to the front and is highlighted in a variety of ways to focus the user's attention. When the user activates a window, the underlying platform

generates the appropriate activate and deactivate events. The active frame and selection for each window may be saved until it is next active, depending on platform-specific user-interface conventions.

OpenDoc notifies all facets of a window when the window is activated or deactivated.

4.6.2.1.2 Zooming Windows

On platforms where window zooming is supported, when the user takes the appropriate action to zoom the window, the document shell can handle the event. The root part should intercept this event, however, and define the appropriate window size.

Note that the new window size might trigger a frame negotiation, and may also require the root part to create or delete facets.

4.6.2.1.3 Moving Windows

The document shell handles any platform-specific window-moving actions. No event-handling is required of the window's root part. If the window moves between monitors of different bit-depth, portions of it may need to be redrawn.

Parts in other windows may need to be updated because of the window's move; they receive update events as appropriate.

4.6.2.1.4 Resizing a Window

The document shell is responsible for resizing all windows, although the root part can intercept and handle the events.

When a window is resized, the active part does not change, but the part editor for the root frame is informed of the resizing, through calls to its `geometry_changed` method. The root part can then do any necessary invalidation and subsequent redrawing, including creation of new facets if embedded parts have become visible due to the resizing.

4.6.2.1.5 Dragging a Window

The document shell typically handles dragging of a window by its title bar, although the root part can intercept and handle this event, if it needs to constrain the movement of the window for some reason.

4.6.2.1.6 Closing a Window or Document

The document shell typically handles the Close menu item or a mouse click in the close box of a window. The document shell closes the window, after which it cannot be reopened. However, the OpenDoc dispatcher first sends a `evt_window` event to the root part. If a part editor wishes to merely hide its window rather than close it (for example, if it is a palette), the part editor can act on this event.

If the window is a document window and is the only one open for that document, the document shell closes the document. The root part can, if desired, intercept the window event and handle the document- closing action itself.

If a part editor needs to close a window programmatically, it can call the window's `close_and_remove` method. The window is closed and the window object deleted.

4.6.2.2 Modal Dialog Boxes

When a part editor displays a modal dialog box, including one provided by the system, it may not be able to create an `Window` object, as with a regular window. However, it should still request the modal focus, and it can still provide a dialog filter to control the events it receives. In addition, to ensure that floating windows are properly deactivated, a part must deactivate the front window before displaying a modal dialog, and reactivate the front window after dismissing it.

4.6.2.2.1 Acquiring and Relinquishing the Modal Focus

The **modal focus** is the focus type that should be possessed by frames that are displaying modal dialog boxes.

The modal focus exists in order to constrain certain events. For example, a mouse click outside the frame with the modal focus will be sent to the modal focus frame, but a click in an embedded frame within the modal focus frame will go to the embedded frame.

A part obtains and relinquishes the modal focus in the same way as other foci, as described in the sections "Requesting Foci" on page 93 and "Relinquishing Foci" on page 93.

A part should in general be unwilling to relinquish the modal focus on request; if it is displaying a modal dialog, it probably does not want any other modal dialog to be displayed at the same time. To make sure that it holds on to the modal focus, a part editor's `begin_relinquish_focus` method should return `FALSE` if the requested focus is `modal_focus` and the proposed new owner of the focus is not one of its own display frames.

4.6.2.2.2 Dialog Filters

With modal dialog boxes, s part editor's dialog-box event filter controls which events it receives while a dialog box or alert box is being displayed. To pass null events, update events, and activate events on to OpenDoc or other windows to handle, its event filter can send them to the OpenDoc dispatcher, by calling its `dispatch` method.

4.6.2.2.3 Handling a Simple Modal Dialog

To display a simple modal dialog box, take these steps:

1. Get a reference to the frame that is the current owner of the modal focus, by calling the arbitrator's `acquire_focus_owner` method.
2. Request the modal focus from the arbitrator, using its `request_focus` method. If the focus is obtained, proceed.
3. To properly handle Macintosh floating windows, deactivate the currently active window by calling the window state object's `deactivate_front_windows` method.
4. Handle the dialog box.
5. Reactivate the previously active window (to restore floating windows), by calling the window state's `activate_front_windows` method.
6. Restore the modal focus to its previous owner, by calling the arbitrator's `transfer_focus` method.

By always saving and restoring the owner of the modal focus, a part can use this approach for nested modal dialog boxes, such as when a dialog box is itself built from several embedded parts.

4.6.2.2.4 Handling a Movable Modal Dialog Box

To use a Macintosh movable modal dialog box in OpenDoc, a part creates a window object (`Window`) to contain it. To display the dialog box, take these steps:

1. Use platform-specific methods to create the structures for the dialog box.
2. Create a window object, using the window state's `register_window` method. Give it the appropriate properties for the modal dialog, such as nonpersistent.
3. Request the modal focus for the root frame of the dialog window, or the frame that is to be active upon opening.
4. Adjust menus as necessary for the presence of the dialog box.
5. Open, show, and select the modal dialog window.
   To make sure that the movable modal dialog box is dismissed at the right time, take actions such as these when a mouse-down event is received in the dialog box:

1. Make any necessary platform-specific calls to handle the event.
2. Get a reference to the dialog box's window from its ID. If the view type of the window's frame corresponds to the part's movable-modal view type, and if the results of the platform-specific calls signify that the user chose to close the dialog box, call the window's `close_and_remove` method to delete the window and its root frame.
3. Re-enable any menus or menu items disabled during display of the dialog box.

4.6.2.3 Modeless Dialog Boxes

Modeless dialog boxes are more like regular windows than are modal dialogs, in that they can be activated and deactivated, and need not go away when a part is active and being edited.

4.6.2.3.1 Showing the Dialog Box

Suppose the user chooses a menu item to show a modeless dialog box. A part may do something like the following:

1. Create a new reference to the dialog-box window from its ID.
   - If the window already exists, get a reference to it by passing its ID to the window state's `get_window` method, and then show and select the window to make it visible and active.
   - If it does not yet exist, create the platform-specific structures for the dialog box, and create a window object with the window state's `create_window` method.

2. Open, show, and select the modeless dialog window to make it visible and active.
3. If the part does not already have it, get and save the ID of the window, through the window's `get_id` method, to use in step 1 the next time the user chooses the modeless dialog box.

4.6.2.3.2 Closing the Dialog Box

When the user clicks in the close box of a modeless dialog, a part may hide the dialog window rather than close it, so that it is not destroyed. In the part's `handle_event` method, respond in this way to a mouse click within a window's go-away box:

1. Get a reference to the window object in which the event occurred, by calling the facet's `get_window` method.
2. Get a reference to the modal dialog's window object, by passing its ID to the window state's `get_window` method.
3. If the two are the same, hide the window instead of closing it.

A part could also make use of the view type of the root frame to identify the window as a modeless dialog box, if you have defined such a view type.

4.6.2.3.3 Hiding a Dialog When Deactivating a Frame

A part should hide any of its modeless dialog boxes when deactivated.

When a part relinquishes the selection focus, it can get a reference to the dialog window (by passing its ID to the window state's `get_window` method), call the window's `is_shown` method to see if it is currently being shown, and then save that shown state and hide the window.

When a part reacquires the selection focus, it can retrieve a reference to the dialog window (by passing its ID to the window state's `get_window` method), and then show the window if it had in a shown state at deactivation.

4.6.3 Controls

This section discusses the different kinds of controls that can be used and the different ways that they can be constructed them, as well as how to handle events within controls.

4.6.3.1 Design Issues for Controls

Controls in OpenDoc fulfill the same function as in conventional applications: they are graphical objects that allow the user to interact with and manipulate documents in a variety of visual ways. However, there are some differences:

- In a conventional application, controls typically apply to the entire document, and so are always present or must be explicitly dismissed. In an OpenDoc document, however, each kind of part may have its own set of controls, and thus controls can appear and disappear rapidly as the user edits. This can be irritating if not carefully managed.
- In a conventional application, finding the space in which to display a control may be less of a consideration than in OpenDoc. Displaying rulers, scroll bars, and palettes for small embedded parts may be challenging.
- In OpenDoc, controls can be constructed as independent parts, assemblages of parts, different frames of a single part, or content elements of a part. This can give more flexibility in constructing user-interface elements than is possible with standard controls.
- OpenDoc controls can have attached scripts or can communicate with each other, or with other parts, using the OpenDoc extension mechanism. This gives OpenDoc controls the ability to be far more integrated, and context-sensitive than standard controls.

Standard types of controls that a part editor might use include:

- push buttons, radio buttons and check boxes
- scroll bars and sliders, or other gauges
- pop-up menus
- rulers
- tool bars
- status bars
- palettes

Rulers usually reflect some settings of the current selection context, and contain controls that allow the user to change these settings. Tool bars are like rulers, but often trigger actions rather then change settings. Status bars display the progress of some long-running operation or suggest actions to users.

In conventional applications, rulers, tool bars, and status bars commonly occupy the margins of the window. In an OpenDoc document, controls can be displayed within the frame they apply to, or in separate frames outside of the frame they apply to.

A ruler for a text part, for example, can be an additional frame associated with the part. Events in the ruler are handled by the text editor associated with the text part. Conversely, the ruler may be its own part with its own editor. In this case the text part editor must maintain a reference to the ruler part and be able to communicate with it, through semantic events or some other extension mechanism.

A ruler that is its own part can have other parts, such as buttons, embedded within it. The ruler part must then of course be able to communicate with its embedded controls as well as with the part that it services.

Palettes often contain editing tools, but can also contain choices for object attributes like color or line width. Palettes commonly float freely beside or over the document, although they can also be fixed at the window margins. Palettes might also pop-up, pull-down, or be torn off of menus.

If all parts in a document can communicate with each other, they can coordinate the drawing and hiding of palettes or other controls, to avoid irritating the user. For example, all parts in a document can share a single palette, within which the various parts negotiate for space, and in which they draw only those tools that must change as the active part changes.

4.6.3.2 Handling Events in Controls

How events in a control are handled depends on how the control is designed:

- If a control like a ruler is within the active frame, it might not have its own frame, and the active part editor handles any events in the ruler.
- If a control has its own frame, it might be another display frame of the active part. in this case also, the active part editor handles the event in the control.
- If a control is a separate part, its own part editor handles the event and updates the state of the part, which might trigger scripts or calls to other parts' extension interfaces. Likewise, the part itself can receive can receive queries from other part editors, in the form of semantic events or calls to its extension interface.

4.6.4 Menus and Menu Commands

At any given moment while an OpenDoc document is open, responsibility for the menu bar is shared among three entities. The operating system provides any system-wide menus, the OpenDoc document shell creates the Document menu and the Edit menu, and individual part editors can create other menus as needed. (Part editors can also, with specific restrictions, add appropriate items to the Document and Edit menus.)

Different platforms have different conventions for enabling and disabling menus and menu items. In an OpenDoc document, the currently active part (specifically, the part with the menu focus) controls which menu commands are available to the user. As each part becomes active, the OpenDoc document shell updates its own menu items, while the active part editor takes care of the rest.

When the user chooses a menu item, the document shell either handles the command itself, or dispatches a menu event to the active part by calling the part's `handle_event` method.

This section discusses general issues of setting up and working with menus, and then describes how to handle menu events for the standard OpenDoc menus (the Document menu and the Edit menu).

4.6.4.1 Working With Menus

This section describes how your part editor can set up and use menus and menu items.

4.6.4.1.1 The Base Menu Bar

When it first opens a document, the document shell creates a menu bar object (type `MenuBar`) and installs it as the base menu bar, by calling the window state's `set_base_menu_bar` method.The base menu bar contains different menus and items on different platforms, but on the Macintosh the Document menu and the Edit menu are always installed.

4.6.4.1.2 Adding Part Menus to the Base Menu Bar

When your part initializes itself, or when it first obtains the menu focus, it should create its own menu bar object, by

1. copying the base menu bar using the window state's `copy_base_menu_bar` method
2. adding its own menu structures

If absolutely necessary, a part editor can add items to the end of the Document and Edit menus, but you should not alter the existing items.

4.6.4.1.3 Registering Command IDs

Part editors can register a command ID for each menu and item, using the menu bar object's `register_command` method. This capability is designed for use on platforms that do not have position-independent menu-item IDs. For better localizability, a part editor should capture the mapping from menu items to command IDs and save it in a resource.

The OpenDoc document shell registers command IDs for all items in the standard menus for each platform, including the Document and Edit menus. If, on a platform without position-independent menu-item IDs, a part editor reorganizes those menus—which is not recommended—it needs to re-register the command numbers so that the document shell can still handle menu items correctly.

4.6.4.1.4 Claiming the Menu Bar

When a part is activated, it should request the menu focus (along with other foci), if it wants to use menus. See "Requesting Foci" on page 93 for more information.

Once a part has the menu focus, it should call the menu bar's `display` method to make its menu bar visible and active.

4.6.4.1.5 Retrieving Command IDs

On platforms without position-independent menu-item IDs, the menu event dispatched to a part editor contains a menu-number/item-number pair that together determine which command was selected by the user. To convert that information to a command ID before dispatching it to the individual menu-item editors, call the `get_Command` method of the menu bar object.

4.6.4.1.6 Enabling and Disabling Menus

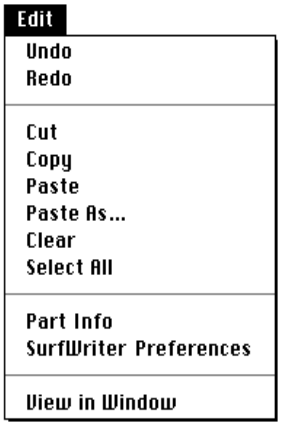
When the mouse is clicked in the menu bar, the OpenDoc dispatcher determines which part has the menu focus, and calls that part's `adjust_menus` method.

A part's `adjust_menus` method can use methods of the menu bar object to change the appearance of its menu items, or it can make platform-specific calls to directly enable, disable, mark, or change the text of its menu items.

4.6.4.2 The Edit Menu

Most items in the Edit menu are handled by individual part editors. Most apply to the current selection in the currently active part. See Figure 47.

Figure 47 The Edit menu



4.6.4.2.1 Undo, Redo

The user selects the Undo or Redo commands from the Edit menu in order to reverse the actions of recently enacted commands, including previous Undo or Redo commands.

The document shell handles the Undo and Redo items, passing control to the `Undo` object. The `Undo` object, in turn, calls the `undo_action` or `redo_action` methods of any part editors involved in the Undo or Redo.

4.6.4.2.2 Cut, Copy, Paste

The user selects the Cut, Copy, or Paste commands from the Edit menu in order to place data onto the Clipboard or retrieve data from the Clipboard.

A part should handle these commands as described in the section “Clipboard Transfer” on page 47.

4.6.4.2.3 Paste As

The user selects the Paste As command from the Edit menu in order to specify the manner in which Clipboard data is to be pasted into the active part. When the user selects the command, OpenDoc passes the command information to a part’s `handle_event` method.

Routines that handle the Paste As command should prepare to read from the Clipboard:

1. Lock the Clipboard and gain access to its root storage unit.
2. Bring up the Paste As dialog box, by calling the `show_paste_as_dialog` method of the Clipboard object. Pass the function the active frame into which the paste is to occur.
3. If the function returns `TRUE`, the user has pressed the OK button; use the results of the interaction to determine which kind of pasting action to take. Then read the appropriate kind of data from the Clipboard.

4.6.4.2.3.1 Drag-and-Drop

A part editor’s `drop` method can also follow this procedure when the state of the modifier keys indicates that the Paste As dialog box should be displayed. (The drag-and-drop object also has a `show_paste_as_dialog` method.)

4.6.4.2.4 Clear

The user selects the Clear command from the Edit menu in order to delete the selected content from the active part.

Routines that handle the Clear command should remove the items that make up the selection from the part content. That may involve deleting embedded parts as well as intrinsic content.

4.6.4.2.5 Select All

The user selects the Select All command from the Edit menu in order to make the current selection encompass all of the content of the active part.

Routines that handle the Select All command needs to include all of the content in the selection structure that the part maintains, and highlight the visible parts of it appropriately.

4.6.4.2.6 Part Info/Link Info

The user selects the Part Info command from the Edit menu in order to display a dialog box containing standard information about the selected part, link, or intrinsic content. When the user selects the command, OpenDoc passes the command information to a part's `handle_event` method. Routine that handle the part Info command should bring up a dialog box that describes the character of the current selection.

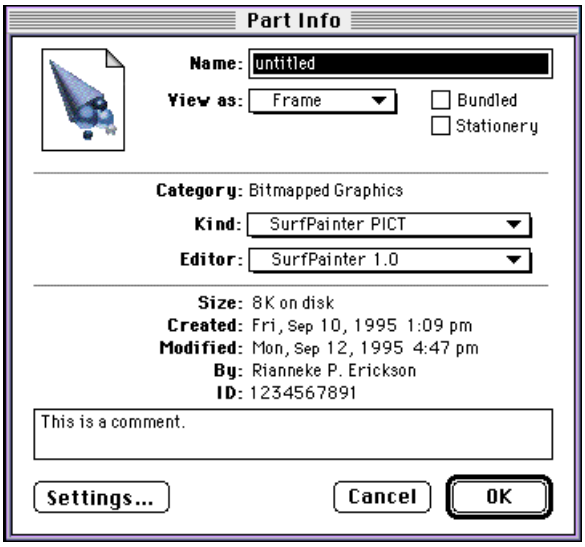
Even before this item is selected, a part needs to make sure that it contains the correct text. Whenever a part is made active, and whenever the selection changes, the `adjust_menus` method should update the name of this menu item as follows:

- If the selection consists of an embedded frame, set the name of the menu item to "Part Info…".
- If the selection consists of a link border, set the name of the menu item to "Link Info…".
- If the selection contains intrinsic content of a part, and if it supports an Info dialog of its own, set the name of the menu item to "<mycontent> Info…", where "<mycontent>" is a brief, meaningful word or phrase describing the selection content.
- If there is no selection or insertion point, set the name of the menu item to "<mypartkind> Info…", where "<mypartkind>" is the part kind of the data the part is displaying.

Routines that handle the Part Info command, can take the following steps. In general, determine what the current selection is, and act accordingly:

- If the current selection is an embedded frame border, bring up the OpenDoc Part Info dialog box for the part in that frame, using the `show_part_frame_info` method of the info object (class `Info`), which is obtained by calling the session object's `get_Info` method. Figure 48 shows an example of the dialog box. OpenDoc handles all changes made by the user and updates the information in the storage units for the embedded part and frame.

Figure 48 The Part Info dialog box

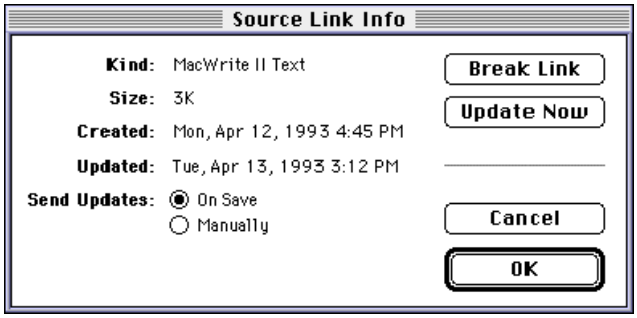


- If a part supports linking and the current selection is the border of a link source, bring up the OpenDoc Link-Source Info dialog box for the link source, using the `show_link_source_info` method of the link source object associated with the selection. Figure 49 shows an example of the dialog box. Handle the results of the function like this:
  - If the user chose to break the link, set the source part to null by calling the link source's `set_source_part` method, and then delete the reference to the link source by calling its override of its

inherited `release` method. Call the `set_changed_from_prev` method of the draft, to make sure that this change is saved when the document is closed.

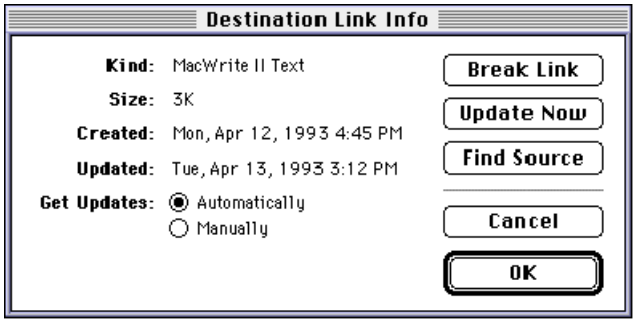
- If the user chooses to update the link immediately, change the auto-update status of the link if necessary (by calling the link source's `set_auto_update` method) and then update the link source from the part's content.
- If the user chooses not to update immediately but has changed the auto-update setting to specify automatic updating, change the auto-update status of the link. Then obtain a new change ID from the link source (by calling its `get_change_id` method) and, if it is different from the currently stored change ID, store the new ID and update the link source.

Figure 49 The Link-Source Info dialog box



- If a part supports linking and the current selection is the border of a link destination, bring up the OpenDoc Link-Destination Info dialog box for the link destination, using the `show_link_destination_info` method of the link object associated with the destination. Figure 50 shows an example of the dialog box. Handle the results of the function like this:
  - From the part's own data structures, obtain a reference to the destination's link object and a pointer to the link info structure associated with it.
  - If the user chooses to break the link, call the link object's `unregister_dependent` method, so that the part will no longer be notified when updates occur, and then delete the reference to the link by calling its override of its inherited `release` method. Call the `set_changed_from_prev` method of the draft, to make sure that this change is saved when the document is closed.
  - If the user has changed the auto-update setting of the link, update the link info structure accordingly. If the change is to specify automatic updating, call the link object's `register_dependent` method, so that the part will be notified when updates occur.
  - If the user chooses to view the source of the link, call the link object's `show_source_content` method.
  - If the user chooses to update the link immediately, update the part's content from the link data.

Figure 50 The Link-Destination Info dialog box



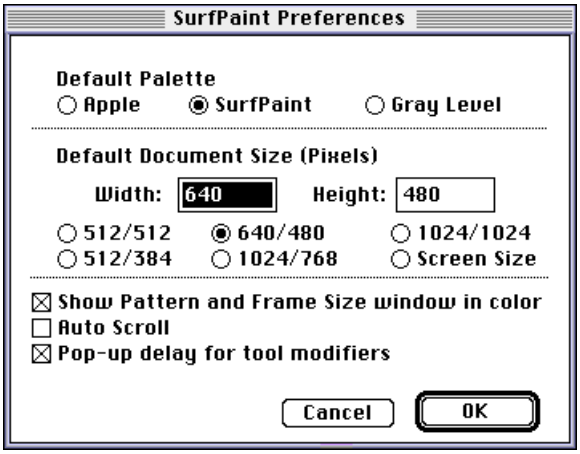
- If the current selection is a portion of a part's intrinsic content and the part supports one or more Content Info dialog boxes of its own, it can display the appropriate dialog box and handle user selections.
- If there is no current selection, or if the selection is intrinsic content and s part does not support its own Content Info dialog box, it must at least display the standard Part Info dialog box for its own part. Get a reference to the part's (active) display frame, by calling the arbitrator's `acquire_focus_owner` method, specifying that it wants the frame that currently owns the menu focus. Then call the `show_part_frame_info` method of the info object for that frame.

4.6.4.2.7 [Active Editor] Preferences

The user selects the [Active Editor] Preferences command from the Edit menu in order to bring up the **editor-preferences dialog box** (Figure 51) in which the user can view and change preferences for the active part's editor.

Even before this item is selected, a part needs to make sure that it contains the correct text. Whenever the part is made active, the `adjust_menus` method should update the name of this menu item to "<myEditor> Preferences", where "<myEditor>" is the name of the part editor.

Figure 51 An example editor-preferences dialog box



Routines that handle the [Active Editor] Preferences command should bring up a dialog box whose contents show the global settings the user can make that affect all of the part editor's parts.

4.6.4.2.8 View in Window

The user selects the View in Window command from the Edit menu in order to open the active part, currently displayed in a frame or as an icon, into its own part window. When the user selects the command, OpenDoc passes the command information to a part's `handle_event` method.

Routines that handle the View in Window menu command should, for each selected frame (in case more than one is selected when the command is given), call the frame's `acquire_part` method to get the part that owns the frame, and then call the `open` method of that part.

5. Implementation model

OpenDoc is designed to be a cross-platform architecture, easily adaptable to any platform both in terms of run time 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, boundaries between executables are expressed in terms of objects and methods.
- There is a persistent storage system with stream based I/O available.

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.

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

- OpenDoc objects do not expose data members to outside view or manipulation.
- Groups of objects are defined 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.

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 and layout. However, it’s quite likely that others will wish to extend the interaction space by adding new methods to the classes involved. This is formalized 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.

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.

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, and HI Events.

Appendix A - Resolution of Requirements

Common facility interfaces shall be object-oriented and shall be expressed in IDL

Any part of a facility that is viewable as an object shall be viewed that way and given an IDL interface description.

Any requirements that a facility places on general objects shall be defined in IDL. For example, a presentation facility might define what a “presentable” object is.

The facility shall use IDL naming conventions for interfaces, types, operations, attributes, etc.

The facility shall use IDL conventions for exceptions and exception parameters.

> All parts of the facility interfaces are object-oriented and expressed in IDL. The IDL is given in appendices B, C, D, E, and F. All parts of the facilities that are viewable as objects are viewed that way and given an IDL interface description. All requirements that the facilities place on general objects are defined in IDL. IDL naming conventions are used throughout.

Proposed extensions to IDL, CORBA, Object Services, and/or the OMG Object Model shall be identified

Common Facilities shall be consistent with IDL, CORBA, Object Services, and the OMG Object Model. It is a goal to minimize extensions required to the CORBA and define additional components that use the CORBA wherever possible.

Moreover, Common Facilities should assume that there exist good implementations of the CORBA specification. They must not work-around early CORBA implementations if this will sacrifice uniformity and generality in the long term.

> No extensions are proposed by the facilities.

Operation sequencing shall be included where applicable

Behavior and sequencing of operations shall be part of that facility’s specifications. (e.g. transactions)

> Behavior and sequencing of operations are provided in general in Section 4., “Functional Overview,” and specifically in the descriptions of the classes and methods in Volume II of this submission.

Common facility specifications shall not contain implementation descriptions

Maximum implementation flexibility should be preserved. However, provision should be made for application developers to control implementation choices (e.g., by pragmas or in some other way that does not affect the Common Facility’s functional interface).

> This submission does not contain implementation descriptions except to provide examples of the semantics and sequencing to permit the use of the facilities

Specifications shall be complete

There shall be no “magic” required. For example, object creation does not just happen, some operation must be called to make it occur. Lifecycle Service’s definitions shall be defined as appropriate to provide these operations (See COSS, Volume 1, Lifecycle Service).

> This submission is a complete specification. No “magic” is required on the part of clients.

Common facilities should have precise descriptions

Common Facilities should have a precise description of their semantics and side-effects, if any. For example, a Common Facility should distinguish interfaces and behaviors provided by a facility from those it expects other Common Facilities to supply.

> This information is provided in the descriptions of the classes and methods in Volume II of this submission.

Independence and modularity of Common Facilities

It should be possible to separately specify and implement each Common Facility.

(It may also be possible to view an existing software module as implementing a collection of one or more OMG Common Facilities.)

Each facility is separately specified and can be implemented separately. Note, however, that the Compound Interchange and Compound Presentation facilities expect the Compound Core and Compound Utilities to be available.

Minimize duplication of functionality (the Bauhaus principle)

Functionality should belong to the most appropriate facility.

Each facility should build on previous facilities where appropriate. For example, if the presentation facility defines object presentation interfaces, an application should be able to use this facility and should not need to reinvent presentation for itself.

The functionality in this submission is partitioned into the most appropriate facilities.

The presence of the Common Core and Common Utility facilities in addition to the Compound Presentation and Compound Interchange facilities is an example of the minimization of duplication. These features of these additional facilities support common dependencies of the Compound Presentation and Interchange facilities. These additional facilities permit the reuse of common functionality, and prevent duplication in other facilities. Note that the additional facilities are defined here because no other common facilities have been adopted to date, so there is no base upon which to build the Compound Presentation and Interchange facilities.

No hidden interfaces among Common Facilities

Interfaces and behaviors must be specified sufficiently well so that implementations can be replaced and the parts will still work together. For example, the operation PutInside(container, member) works independently of the implementation of members (via a database, ORB, etc.).

There are no hidden interfaces. The interfaces and behaviors in Volume II of this submission meet the above criteria.

Consistency among Common Facilities

Different Common Facilities must be able to work together.

The facilities are defined in a matter that meets the above criteria.

The technology defined here is based on implementations working on many disparate vendor platforms.

Extensibility of individual Common Facilities

Common facilities should be extensible through an iterative process. It should be clear how extensions can be defined (e.g., via inheritance, delegation, etc.).

Common Facilities should use the multiple inheritance capability of IDL wherever possible. Multiple inheritance makes it easier to define a set of interfaces that can be used by different facilities. A sub-interface can inherit from several different interfaces and use multiple inheritance to specify the set of operations it implements. Clients may choose to use only part of the behavior of any object, and view the object with just the interface they need

The Compound Core facility that is used by the Compound Presentation and Compound Interchange facilities includes an extension interface.

Extending the collection of Common Facilities

It should be possible to define and standardize new Common Facilities without having to redesign a system that uses existing Common Facilities.

Nothing in the facilities defined in this submission depend on facilities defined outside of this submission. As a result, the definition of new Common Facilities will not affect the design of the facilities defined in this submission. However, the facilities defined in this submission include a number of conventions such as those for nam-

ing; new Common Facilities must adhere to these conventions.

Configurability

It should be possible to configure Common Facilities in different combinations for different purposes. It should be possible to use two or more facilities in arbitrary combinations.

The facilities included in this submission can be configured in arbitrary configurations. However, not all combinations are useful.

The Compound Presentation and Compound Interchange facilities can be configured separately or together as long as the Compound Core and Compound Utility facilities are available.

Integrity, Reliability, and Safety of Common Facilities.

Safeguards should be provided to guard against corrupting Common Facilities and the objects they manage.

OpenDoc provides features to facilitate the integrity, reliability and safety of presentation and interchange up to a point. Here are some examples:

- different types of clipping are provided, and may be used at different times, to aid display integrity.
- applications can provide multiple data models to aid reliable interchange of data even when the applications are unknown to one another.
- built-in versioning and undoability provide two levels of data safety.

OpenDoc relies on typically available system features, such as file security and recoverability, for other levels of integrity, reliability and safety. However, OpenDoc was not designed to provide guarantees in these areas. The submitters nevertheless believe that a properly written OpenDoc program will not damage data or the behavior of other applications.

Performance.

Interfaces to Common Facilities should be designed with performance trade-offs in mind.

OpenDoc interfaces were designed with performance tradeoffs in mind. If requested by the Common Facilities Task Force, specific examples can be provided. Current implementation efforts on a number of different platforms have not revealed any significant performance penalties.

Scalability.

Common Facilities should be specified with the goal that there may be many implementations that are optimized for different environments. OMG IDL and a language mapping can make remote facilities look like object invocations. Most Common Facilities have a possible implementation across a spectrum from a shared library to a remote server. The user of a Common Facility should not have to care which variety is being used.

The facilities included in this submission have been designed to scale across a wide range of environments including personal computer operating systems, multi-tasking virtual memory workstation-class operating systems, and networked systems.

Scalability is enhanced by the reliance on a CORBA-compliant ORB, and the use of other object services. Details such as message transmission and cross-platform interoperation are handled at a level below that which the developer using these facilities sees.

Portability.

Common Facilities should be designed to accommodate portability of implementations across a wide range of platforms. They should not require use of a particular programming language.

The definition of the facilities in IDL ensures that implementations will be portable across a wide range of plat-

forms and programming languages. Note, however, that not all aspects of applications are addressed by these facilities. Therefore, some portions of the application source code will need to be modified.

In all cases, the non-portable features depend on facilities, such as rendering, for which no applicable standard currently exists. Portability will be enhanced as such facilities are standardized.

Consistency with Common Object Services Specifications (COSS)

Consistency and compliance with COSS, Volume 1 (and other COSS specifications, if they exist at the time of submission) wherever compliance is relevant and appropriate, including:

Provision of Lifecycle Object's Factory Service as needed (see COSS, Volume 1) to use the service or create objects managed by the service including the definition of all factory interfaces.

This submission uses COSS where appropriate.

Conventions and guidelines

Conventions and guidelines defined, used or implied in the submission. See Appendix E for a description of why conventions and guidelines are important and for a list of conventions and guidelines that may be applicable to a submissions.

This specification adheres to OMG guidelines and conventions for IDL.

Mandatory versus optional interfaces

Mandatory interfaces that are required to be supported by compliant implementations as distinguished from optional interfaces that a particular implementation may or may not support. Optional interfaces need not be supported for an implementation to be said to be compliant. Complex optional interface schemes are not encouraged.

All interfaces defined by this submission are mandatory.

Constraints on object behavior

Constraints on object behavior over and above those defined by the object's interface semantics (a.k.a. "qualities of service") that implementations must take into account.

The objects defined by these facilities have no such constraints.

Integration with future Common Facilities

How the facility can evolve to integrate with relevant future Common Facilities specifications.

New common facilities can affect the facilities defined in this standard in two ways. First, new facilities that address aspects of applications currently not standardized, such as rendering, can enhance application portability. Second, new facilities can build on the base functionality to provide additional features such as scripting.

Your organization must be an OMG Corporate Member and must provide a statement about your willingness to comply with the OMG's requirements (e.g., your willingness to make the proposed technology commercially available) in your Letter of Intent (see Section 6.1).

This information has been provided in the Letter of Intent that has been submitted.

Submissions must include a "proof of concept" statement, explaining how the submitted specifications have been demonstrated to be technically viable.

It is important for the Task Force to understand the technical viability of an OMG submission during the evaluation process. The technical viability has a lot to do with the state of development of the technology submitted. This is not the same as commercial availability, which is an OMG Board of Directors consideration. Proof of concept statements can contain any information deemed relevant by the submitter, for example:

"This specification has completed the design phase and is the process of being prototyped."

"An implementation of this specification has been in beta-test for 4 months."

“A named product (with a specified customer base) is a realization of this specification.”

Implementations of the basis for this specification have been shipped to customers by multiple vendors on multiple operating platforms.

Appendix B - IDL for the Compound Interchange Facility

See “Appendix F - Types and Constants” for the data type and constant definitions.

Clipboard

```
module CfDoc {

      interface Clipboard :  Object
      {
            UpdateID get_update_id()
                  raises(DocException);

            void clear()
                  raises(DocException);

            StorageUnit get_content_storage_unit()
                  raises(DocException);

            UpdateID action_done(in CloneKind clone_kind)
                  raises(DocException);

            void action_undone(
                  in UpdateID             update,
                  in CloneKind            original_clone_kind
            )
                  raises(DocException);

            void action_redone(
                  in UpdateID              update,
                  in CloneKind            original_clone_kind
            )
                  raises(DocException);

            void set_platform_clipboard(in PlatformTypeList type_list)
                  raises(DocException);

            void export_clipboard()
                  raises(DocException);

            void draft_saved(in Draft draft)
                  raises(DocException);

            void draft_closing(in Draft draft)
                  raises(DocException);
      };
```

```
};
```

Container

```
module CfDoc {

      interface Container :  RefCntObject
      {
            StorageSystem get_storage_system()
                  raises(DocException);

            ContainerID get_id()
                  raises(DocException);

            ContainerName get_name()
                  raises(DocException);

            void set_name(in ContainerName name)
                  raises(DocException);

            Document acquire_document(in DocumentID id)
                  raises(DocException);
      };

};
```

Document

```
module CfDoc {

      interface Document :  RefCntObject
      {
            Container get_container()
                  raises(DocException);

            DocumentID get_id()
                  raises(DocException);

            DocumentName get_name()
                  raises(DocException);

            void set_name(in DocumentName name)
                  raises(DocException);

            Document collapse_drafts(in Draft from, in Draft to)
```

```
            raises(DocException);

        Draft acquire_draft(
            in DraftPermissions     perms,
            in DraftID              id,
            in Draft                draft,
            in PositionCode         pos_code,
            in boolean              release
        )
            raises(DocException);

        boolean exists(
            in DraftID              id,
            in Draft                draft,
            in PositionCode         pos_code
        )
            raises(DocException);

        Draft acquire_base_draft(in DraftPermissions perms)
            raises(DocException);

        Draft create_draft(in Draft below, in boolean release_below)
            raises(DocException);

        void save_to_a_prev_draft(in Draft from, in Draft to)
            raises(DocException);

        void set_base_draft_from_foreign_draft(in Draft draft)
            raises(DocException);
    };

};
```

Draft

```
module CfDoc {

    interface Draft :  RefCntObject
    {
        Document get_document()
            raises(DocException);

        DraftID get_id()
            raises(DocException);

        StorageUnit acquire_draft_properties()
```

```
      raises(DocException);

DraftPermissions get_permissions()
      raises(DocException);

StorageUnit create_storage_unit()
      raises(DocException);

StorageUnit acquire_storage_unit(in StorageUnitID id)
      raises(DocException);

void remove_storage_unit(in StorageUnit storage_unit)
      raises(DocException);

boolean is_valid_id(in ID id)
      raises(DocException);

DraftKey begin_clone(
      in Draft               dest_draft,
      in Frame               dest_frame,
      in CloneKind           kind
)
      raises(DocException);

void end_clone(in DraftKey key)
      raises(DocException);

void abort_clone(in DraftKey key)
      raises(DocException);

ID clone(
      in DraftKey            key,
      in ID                  from_object_id,
      in ID                  to_object_id,
      in ID                  scope
)
      raises(DocException);

ID weak_clone(
      in DraftKey            key,
      in ID                  object_id,
      in ID                  to_object_id,
      in ID                  scope
)
      raises(DocException);
```

```
boolean changed_from_prev()
      raises(DocException);

void set_changed_from_prev()
      raises(DocException);

void remove_from_document()
      raises(DocException);

Draft remove_changes()
      raises(DocException);

Draft externalize()
      raises(DocException);

Draft save_to_a_previous(in Draft to)
      raises(DocException);

Frame create_frame(
      in Type                frame_type,
      in Frame               containing_frame,
      in Shape               frame_shape,
      in Canvas              bias_canvas,
      in Part                part,
      in TypeToken           view_type,
      in TypeToken           presentation,
      in boolean             is_subframe,
      in boolean             is_overlaid
)
      raises(DocException);

Frame acquire_frame(in StorageUnitID id)
      raises(DocException);

void remove_frame(in Frame frame)
      raises(DocException);

Part create_part(in Type part_type, in Editor optional_editor)
      raises(DocException);

Part acquire_part(in StorageUnitID id)
      raises(DocException);

void release_part(in Part part)
      raises(DocException);
```

```
            void remove_part(in Part part)
                  raises(DocException);

            LinkSpec create_link_spec(in Part part, in ByteArray data)
                  raises(DocException);

            LinkSource create_link_source(in Part part)
                  raises(DocException);

            LinkSource acquire_link_source(in StorageUnitID id)
                  raises(DocException);

            Link acquire_link(in StorageUnitID id, in LinkSpec link_spec)
                  raises(DocException);

            void remove_link(in Link link)
                  raises(DocException);

            void remove_link_source(in LinkSource link)
                  raises(DocException);

            PersistentObjectID get_persistent_object_id(
                  in PersistentObject     object,
                  in ObjectType           object_type
            )
                  raises(DocException);

            PersistentObject acquire_persistent_object(
                  in PersistentObjectID   object_id,
                  out ObjectType          object_type
            )
                  raises(DocException);
      };

};
```

DragAndDrop

```
module CfDoc {

      interface DragAndDrop :  Object
      {
            void clear()
                  raises(DocException);

            StorageUnit get_content_storage_unit()
```

```
                raises(DocException);

            DropResult start_drag(
                in Frame                src_frame,
                in Type                 image_type,
                in ByteArray            image,
                out Part                dest_part,
                in ByteArray            ref_con
            )
                raises(DocException);
        };

};
```

Link

```
module CfDoc {

        interface Link : PersistentObject
        {
            boolean lock(in unsigned long wait, out LinkKey key)
                raises(DocException);

            void unlock(in LinkKey key)
                raises(DocException);

            StorageUnit get_content_storage_unit(in LinkKey key)
                raises(DocException);

            void register_dependent(in Part client_part, in UpdateID id)
                raises(DocException);

            void unregister_dependent(in Part client_part)
                raises(DocException);

            UpdateID get_update_id()
                raises(DocException);

            Time get_change_time()
                raises(DocException);

            void show_source_content()
                raises(DocException);
        };

};
```

LinkManager

```
module CfDoc {

      interface LinkManager : Object
      {
            boolean unsaved_exported_links(in Draft draft)
                  raises(DocException);

            void draft_opened(in Draft draft)
                  raises(DocException);

            void draft_saved(in Draft draft)
                  raises(DocException);

            void draft_closing(in Draft draft)
                  raises(DocException);

            LinkSource create_link(in Draft draft, in LinkSpec link_spec)
                  raises(DocException);
      };

};
```

LinkSource

```
module CfDoc {

      interface LinkSource : PersistentObject
      {
            boolean lock(in unsigned long wait, out LinkKey key)
                  raises(DocException);

            void unlock(in LinkKey key)
                  raises(DocException);

            void clear(in UpdateID id, in LinkKey key)
                  raises(DocException);

            StorageUnit get_content_storage_unit(in LinkKey key)
                  raises(DocException);

            void content_updated(in UpdateID id, in LinkKey key)
                  raises(DocException);
```

```
            void show_source_content()
                  raises(DocException);

            //#--------------------------------
            //# getters

            UpdateID get_update_id()
                  raises(DocException);

            Time get_change_time()
                  raises(DocException);

            boolean is_auto_update()
                  raises(DocException);

            Link get_link()
                  raises(DocException);

            //#--------------------------------
            //# setters

            void set_auto_update(in boolean automatic)
                  raises(DocException);

            void set_source_part(in StorageUnit source_part_su)
                  raises(DocException);
      };

};
```

LinkSpec

```
module CfDoc {

      interface LinkSpec :  Object
      {
            void write_link_spec(in StorageUnit su)
                  raises(DocException);

            void read_link_spec(in StorageUnit su)
                  raises(DocException);
      };

};
```

StorageSystem

```
module CfDoc {

      interface StorageSystem : Object
      {
            Session get_session()
                  raises(DocException);

            Container acquire_container(
                  in ContainerType        container_type,
                  in ContainerID          id
            )
                  raises(DocException);

            Container create_container(
                  in ContainerType        container_type,
                  in ContainerID          id
            )
                  raises(DocException);

            void need_space(in Size mem_size, in boolean do_purge)
                  raises(DocException);

            TypeList create_type_list(in TypeList type_list)
                  raises(DocException);

            PlatformTypeList create_platform_type_list(in PlatformTypeList
type_list)
                  raises(DocException);
      };

};
```

StorageUnit

```
module CfDoc {

      interface StorageUnit :  RefCntObject
      {
            Draft get_draft()
                  raises(DocException);

            Session get_session()
                  raises(DocException);
```

```
boolean exists(
      in PropertyName         property_name,
      in ValueType            value_type,
      in ValueIndex           value_index
)
      raises(DocException);

boolean exists_with_cursor(in StorageUnitCursor cursor)
      raises(DocException);

unsigned long count_properties()
      raises(DocException);

unsigned long count_values()
      raises(DocException);

StorageUnit focus(
      in PropertyName         property_name,
      in PositionCode         property_pos_code,
      in ValueType            value_type,
      in ValueIndex           value_index,
      in PositionCode         value_pos_code
)
      raises(DocException);

StorageUnit focus_with_cursor(in StorageUnitCursor cursor)
      raises(DocException);

StorageUnit externalize()
      raises(DocException);

StorageUnit internalize()
      raises(DocException);

ID get_id()
      raises(DocException);

StorageUnitName get_name()
      raises(DocException);

void set_name(in StorageUnitName name)
      raises(DocException);

StorageUnit add_property(in PropertyName property_name)
      raises(DocException);
```

```
StorageUnit add_value(in ValueType type)
      raises(DocException);

StorageUnit remove()
      raises(DocException);

void clone_into(
      in DraftKey             key,
      in StorageUnit          dest_storage_unit,
      in ID                   scope_id
)
      raises(DocException);

StorageUnitView create_view()
      raises(DocException);

StorageUnitCursor create_cursor_with_focus()
      raises(DocException);

StorageUnitCursor create_cursor(
      in PropertyName         property_name,
      in ValueType            value_type,
      in ValueIndex           value_index
)
      raises(DocException);

PropertyName get_property()
      raises(DocException);

ValueType get_type()
      raises(DocException);

void set_type(in ValueType value_type)
      raises(DocException);

void set_offset(in unsigned long offset)
      raises(DocException);

unsigned long get_offset()
      raises(DocException);

unsigned long get_value(
      in unsigned long        length,
      out ByteArray value
)
      raises(DocException);
```

```
void set_value(in ByteArray value)
      raises(DocException);

void insert_value(in ByteArray value)
      raises(DocException);

void delete_value(in unsigned long length)
      raises(DocException);

unsigned long get_size()
      raises(DocException);

boolean is_valid_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

void get_strong_storage_unit_ref(
      in StorageUnitID        embedded_su_id,
      out StorageUnitRef      strong_ref
)
      raises(DocException);

void get_weak_storage_unit_ref(
      in StorageUnitID        embedded_su_id,
      out StorageUnitRef      weak_ref
)
      raises(DocException);

boolean is_strong_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

boolean is_weak_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnit remove_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnitID get_id_from_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnitRefIterator create_storage_unit_ref_iterator()
      raises(DocException);

void set_storage_unit_ref(
      in StorageUnitID        embedded_su_id,
      in StorageUnitRef       ref
```

```
            )
                  raises(DocException);

            unsigned long get_generation_number()
                  raises(DocException);

            unsigned long increment_generation_number()
                  raises(DocException);

            boolean is_promise_value()
                  raises(DocException);

            void set_promise_value(
                  in ValueType            value_type,
                  in unsigned             long offset,
                  in ByteArray            value,
                  in Part                 source_part
            )
                  raises(DocException);

            unsigned long get_promise_value(
                  in ValueType            value_type,
                  in unsigned             long offset,
                  in unsigned             long length,
                  out ByteArray           value,
                  out Part                source_part
            )
                  raises(DocException);

            void clear_all_promises()
                  raises(DocException);

            void resolve_all_promises()
                  raises(DocException);

            StorageUnitKey lock(in StorageUnitKey key)
                  raises(DocException);

            void unlock(in StorageUnitKey key)
                  raises(DocException);
      };

};
```

StorageUnitCursor

```
module CfDoc {

      interface StorageUnitCursor : Object
      {
            void get_property(out PropertyName property_name)
                  raises(DocException);

            void get_value_type(out ValueType value_type)
                  raises(DocException);

            void get_value_index(out ValueIndex value_index)
                  raises(DocException);

            void set_property(in PropertyName property_name)
                  raises(DocException);

            void set_value_type(in ValueType value_type)
                  raises(DocException);

            void set_value_index(in ValueIndex value_index)
                  raises(DocException);
      };

};
```

StorageUnitView

```
module CfDoc {

      interface StorageUnitView :  Object
      {
            StorageUnit get_storage_unit()
                  raises(DocException);

            StorageUnitCursor get_cursor()
                  raises(DocException);

            StorageUnitView externalize()
                  raises(DocException);

            StorageUnitView internalize()
                  raises(DocException);

            ID get_id()
                  raises(DocException);
```

```
StorageUnitName get_name()
      raises(DocException);

void set_name(in StorageUnitName name)
      raises(DocException);

StorageUnitView add_property(in PropertyName property_name)
      raises(DocException);

StorageUnitView add_value(in ValueType type)
      raises(DocException);

StorageUnitView remove()
      raises(DocException);

void clone_into(
      in DraftKey             key,
      in StorageUnit          dest_storage_unit,
      in ID                   scope_id
)
      raises(DocException);

PropertyName get_property()
      raises(DocException);

ValueType get_type()
      raises(DocException);

void set_type(in ValueType value_type)
      raises(DocException);

void set_offset(in unsigned long offset)
      raises(DocException);

unsigned long get_offset()
      raises(DocException);

unsigned long get_value(
      in unsigned long        length,
      out ByteArray           value
)
      raises(DocException);

void set_value(in ByteArray value)
      raises(DocException);
```

```
void insert_value(in ByteArray value)
      raises(DocException);

void delete_value(in unsigned long length)
      raises(DocException);

unsigned long get_size()
      raises(DocException);

boolean is_valid_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

void get_strong_storage_unit_ref(
      in StorageUnitID        embedded_su_id,
      out StorageUnitRef      strong_ref
)
      raises(DocException);

void get_weak_storage_unit_ref(
      in StorageUnitID        embedded_su_id,
      out StorageUnitRef      weak_ref
)
      raises(DocException);

boolean is_strong_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

boolean is_weak_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnitView remove_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnitID get_id_from_storage_unit_ref(in StorageUnitRef ref)
      raises(DocException);

StorageUnitRefIterator create_storage_unit_ref_iterator()
      raises(DocException);

unsigned long get_generation_number()
      raises(DocException);

unsigned long increment_generation_number()
      raises(DocException);

boolean is_promise_value()
```

```
                raises(DocException);

            void set_promise_value(
                in ValueType            value_type,
                in unsigned             long offset,
                in ByteArray            value,
                in Part                 source_part
            )
                raises(DocException);

            unsigned long get_promise_value(
                in ValueType            value_type,
                in unsigned             long offset,
                in unsigned             long length,
                out ByteArray           value,
                out Part                source_part
            )
                raises(DocException);
        };
};
```

Appendix C - IDL for the Compound Presentation Facility

See “Appendix F - Types and Constants” for the data type and constant definitions.

Arbitrator

```
module CfDoc {

      interface Arbitrator :  Object
      {
            boolean request_focus_set(
                  in FocusSet             focus_set,
                  in Frame                requesting_frame
            )
                  raises(DocException);

            boolean request_focus(
                  in TypeToken            focus,
                  in Frame                requesting_frame
            )
                  raises(DocException);

            void relinquish_focus_set(
                  in FocusSet             focus_set,
                  in Frame                relinquishing_frame
            )
                  raises(DocException);

            void relinquish_focus(
                  in TypeToken            focus,
                  in Frame                relinquishing_frame
            )
                  raises(DocException);

            void transfer_focus(
                  in TypeToken            focus,
                  in Frame                transferring_frame,
                  in Frame                new_owner
            )
                  raises(DocException);

            void transfer_focus_set(
                  in FocusSet             focus_set,
                  in Frame                transferring_frame,
                  in Frame                new_owner
            )
```

```
            raises(DocException);

        Frame acquire_focus_owner(in TypeToken focus)
            raises(DocException);

        FocusOwnerIterator create_owner_iterator(in TypeToken focus)
            raises(DocException);

        void register_focus(
            in TypeToken            focus,
            in FocusModule focus_module
        )
            raises(DocException);

        void unregister_focus(in TypeToken focus)
            raises(DocException);

        boolean is_focus_registered(in TypeToken focus)
            raises(DocException);

        boolean is_focus_exclusive(in TypeToken focus)
            raises(DocException);

        FocusModule get_focus_module(in TypeToken focus)
            raises(DocException);

        FocusSet create_focus_set()
            raises(DocException);
    };

};
```

Canvas

```
module CfDoc {

    interface Canvas :  Object
    {
        boolean has_platform_canvas(in GraphicsSystem g)
            raises(DocException);

        PlatformCanvas get_platform_canvas(in GraphicsSystem g)
            raises(DocException);

        void set_platform_canvas(in GraphicsSystem g, in PlatformCanvas c)
            raises(DocException);
```

```
boolean has_platform_print_job(in GraphicsSystem g)
      raises(DocException);

PlatformPrintJob get_platform_print_job(in GraphicsSystem g)
      raises(DocException);

void set_platform_print_job(
      in GraphicsSystem       g,
      in PlatformPrintJob     j
)
      raises(DocException);

Part acquire_owner()
      raises(DocException);

void set_owner(in Part owner)
      raises(DocException);

Facet get_facet()
      raises(DocException);

void set_facet(in Facet facet)
      raises(DocException);

Transform acquire_bias_transform()
      raises(DocException);

void set_bias_transform(in Transform x)
      raises(DocException);

boolean is_dynamic()
      raises(DocException);

boolean is_offscreen()
      raises(DocException);

Shape acquire_update_shape()
      raises(DocException);

void reset_update_shape()
      raises(DocException);

void invalidate(in Shape shape)
      raises(DocException);
```

```
            void validate(in Shape shape)
                  raises(DocException);
      };

};
```

Dispatcher

```
module CfDoc {

      interface Dispatcher :  Object
      {
            void add_dispatch_module(
                  in EventType            event_type,
                  in DispatchModule       dispatch_module
            )
                  raises(DocException);

            DispatchModule get_dispatch_module(in EventType event_type)
                  raises(DocException);

            void remove_dispatch_module(in EventType event_type)
                  raises(DocException);

            void add_monitor(
                  in EventType            event_type,
                  in DispatchModule       dispatch_module
            )
                  raises(DocException);

            void remove_monitor(
                  in EventType            event_type,
                  in DispatchModule       dispatch_module
            )
                  raises(DocException);

            boolean dispatch(inout EventData event_data)
                  raises(DocException);

            boolean redispatch(
                  inout EventData         event_data,
                  inout EventInfo         event_info
            )
                  raises(DocException);

            void exit()
```

```
                raises(DocException);

            boolean should_exit()
                raises(DocException);
        };

};
```

DispatchModule

```
module CfDoc {

        interface DispatchModule : Object
        {
            void init_dispatch_module(in Session session)
                raises(DocException);

            boolean dispatch(
                inout EventData         event,
                inout EventInfo         event_info
            )
                raises(DocException);
        };

};
```

Facet

```
module CfDoc {

        interface Facet :  Object
        {
            Frame get_frame()
                raises(DocException);

            Facet create_embedded_facet(
                in Frame                frame,
                in Shape                clip_shape,
                in Transform            external_transform,
                in Canvas               canvas,
                in Canvas               bias_canvas,
                in Facet                sibling_facet,
                in FramePosition        position
            )
                raises(DocException);
```

```
void remove_facet(in Facet facet)
      raises(DocException);

void move_before(in Facet child, in Facet sibling)
      raises(DocException);

void move_behind(in Facet child, in Facet sibling)
      raises(DocException);

Facet get_containing_facet()
      raises(DocException);

FacetIterator create_facet_iterator(
      in TraversalType        traversal_type,
      in SiblingOrder         sibling_order
)
      raises(DocException);

Shape create_shape()
      raises(DocException);

Shape acquire_clip_shape(in Canvas bias_canvas)
      raises(DocException);

void change_geometry(
      in Shape                clip_shape,
      in Transform            transform,
      in Canvas               bias_canvas
)
      raises(DocException);

Shape acquire_aggregate_clip_shape(in Canvas bias_canvas)
      raises(DocException);

Shape acquire_window_aggregate_clip_shape(in Canvas bias_canvas)
      raises(DocException);

Shape acquire_active_shape(in Canvas bias_canvas)
      raises(DocException);

void change_active_shape(
      in Shape                active_shape,
      in Canvas               bias_canvas
)
      raises(DocException);
```

```
Transform create_transform()
      raises(DocException);

Transform acquire_external_transform(in Canvas bias_canvas)
      raises(DocException);

Transform acquire_frame_transform(in Canvas bias_canvas)
      raises(DocException);

Transform acquire_content_transform(in Canvas bias_canvas)
      raises(DocException);

Transform acquire_window_frame_transform(in Canvas bias_canvas)
      raises(DocException);

Transform acquire_window_content_transform(in Canvas bias_canvas)
      raises(DocException);

Canvas create_canvas(
      in GraphicsSystem       graphics_system,
      in PlatformCanvas       platform_canvas,
      in boolean              is_dynamic,
      in boolean              is_offscreen
)
      raises(DocException);

boolean has_canvas()
      raises(DocException);

Canvas get_canvas()
      raises(DocException);

void change_canvas(in Canvas canvas)
      raises(DocException);

Window get_window()
      raises(DocException);

InfoType get_part_info()
      raises(DocException);

void set_part_info(in InfoType part_info)
      raises(DocException);

DragResult drag_enter(
```

```
        in Point                point,
        in DragItemIterator     drag_info,
        in Canvas               bias_canvas
    )
        raises(DocException);

    DragResult drag_within(
        in Point                point,
        in DragItemIterator     drag_info,
        in Canvas               bias_canvas
    )
        raises(DocException);

    void drag_leave(in Point point, in Canvas bias_canvas)
        raises(DocException);

    DropResult drop(
        in Point                point,
        in DragItemIterator     drop_info,
        in Canvas               bias_canvas
    )
        raises(DocException);

    void update(in Shape invalid_shape, in Canvas bias_canvas)
        raises(DocException);

    void invalidate(in Shape invalid_shape, in Canvas bias_canvas)
        raises(DocException);

    void validate(in Shape valid_shape, in Canvas bias_canvas)
        raises(DocException);

    void draw(in Shape invalid_shape, in Canvas bias_canvas)
        raises(DocException);

    void draw_children(in Shape invalid_shape, in Canvas bias_canvas)
        raises(DocException);

    void draw_children_always(
        in Shape                invalid_shape,
        in Canvas               bias_canvas
    )
        raises(DocException);

    void drawn_in(in Shape shape, in Canvas bias_canvas)
        raises(DocException);
```

```
            void invalidate_active_border()
                  raises(DocException);

            void draw_active_border()
                  raises(DocException);

            boolean contains_point(in Point point, in Canvas bias_canvas)
                  raises(DocException);

            boolean active_border_contains_point(
                  in Point                point,
                  in Canvas               bias_canvas
            )
                  raises(DocException);

            boolean is_selected()
                  raises(DocException);

            void set_selected(in boolean is_selected)
                  raises(DocException);

            Highlight get_highlight()
                  raises(DocException);

            void change_highlight(in Highlight highlight)
                  raises(DocException);
      };

};
```

FacetIterator

```
module CfDoc {

      interface FacetIterator : Object
      {
            Facet first()
                  raises(DocException);

            Facet next()
                  raises(DocException);

            void skip_children()
                  raises(DocException);
```

```
            boolean is_not_complete()
                  raises(DocException);
      };

};
```

FocusModule

```
module CfDoc {

      interface FocusModule : Object
      {
            void init_focus_module(in Session session)
                  raises(DocException);

            boolean is_focus_exclusive(in TypeToken focus)
                  raises(DocException);

            void set_focus_ownership(in TypeToken focus, in Frame frame)
                  raises(DocException);

            void unset_focus_ownership(in TypeToken focus, in Frame frame)
                  raises(DocException);

            void transfer_focus_ownership(
                  in TypeToken            focus,
                  in Frame                transferring_frame,
                  in Frame                new_owner
            )
                  raises(DocException);

            Frame acquire_focus_owner(in TypeToken focus)
                  raises(DocException);

            FocusOwnerIterator create_owner_iterator(in TypeToken focus)
                  raises(DocException);

            boolean begin_relinquish_focus(
                  in TypeToken            focus,
                  in Frame                requesting_frame
            )
                  raises(DocException);

            void commit_relinquish_focus(
                  in TypeToken            focus,
                  in Frame                requesting_frame
```

```
            )
                  raises(DocException);

            void abort_relinquish_focus(
                  in TypeToken            focus,
                  in Frame                requesting_frame
            )
                  raises(DocException);
      };

};
```

FocusSet

```
module CfDoc {

      interface FocusSet : Object
      {
            void add(in TypeToken focus)
                  raises(DocException);

            void remove(in TypeToken focus)
                  raises(DocException);

            boolean contains(in TypeToken focus)
                  raises(DocException);

            FocusSetIterator create_iterator()
                  raises(DocException);
      };

};
```

Frame

```
module CfDoc {

      interface Frame :  PersistentObject
      {
            //#--------------------------------
            //# Containing frame

            Frame acquire_containing_frame()
                  raises(DocException);
```

```
void set_containing_frame(in Frame frame)
      raises(DocException);

//#---------------------------------
//# Window

Window acquire_window()
      raises(DocException);

void set_window(in Window window)
      raises(DocException);

//#---------------------------------
//# Getters/setters

unsigned long get_frame_group()
      raises(DocException);

void set_frame_group(in unsigned long group_id)
      raises(DocException);

unsigned long get_sequence_number()
      raises(DocException);

void change_sequence_number(in unsigned long sequence_number)
      raises(DocException);

boolean is_root()
      raises(DocException);

boolean is_subframe()
      raises(DocException);

void set_subframe(in boolean is_subframe)
      raises(DocException);

boolean is_overlaid()
      raises(DocException);

boolean is_frozen()
      raises(DocException);

void set_frozen(in boolean is_frozen)
      raises(DocException);

boolean does_propagate_events()
```

```
      raises(DocException);

void set_propagate_events(in boolean does_propagate_events)
      raises(DocException);

boolean is_in_limbo()
      raises(DocException);

void set_in_limbo(in boolean is_in_limbo)
      raises(DocException);

//#---------------------------------
//# Part

Part acquire_part()
      raises(DocException);

void change_part(in Part part)
      raises(DocException);

InfoType get_part_info()
      raises(DocException);

void set_part_info(in InfoType part_info)
      raises(DocException);

TypeToken get_view_type()
      raises(DocException);

void set_view_type(in TypeToken view_type)
      raises(DocException);

void change_view_type(in TypeToken view_type)
      raises(DocException);

TypeToken get_presentation()
      raises(DocException);

void set_presentation(in TypeToken presentation)
      raises(DocException);

void change_presentation(in TypeToken presentation)
      raises(DocException);

//#---------------------------------
//# Facets
```

```
void facet_added(in Facet facet)
      raises(DocException);

void facet_removed(in Facet facet)
      raises(DocException);

FrameFacetIterator create_facet_iterator()
      raises(DocException);

//#--------------------------------
//# Shapes

Shape create_shape()
      raises(DocException);

//#--------------------------------
//# Frame Shape

Shape acquire_frame_shape(in Canvas bias_canvas)
      raises(DocException);

void change_frame_shape(in Shape shape, in Canvas bias_canvas)
      raises(DocException);

Shape request_frame_shape(in Shape shape, in Canvas bias_canvas)
      raises(DocException);

//#--------------------------------
//# Used Shape

Shape acquire_used_shape(in Canvas bias_canvas)
      raises(DocException);

void change_used_shape(in Shape shape, in Canvas bias_canvas)
      raises(DocException);

//#--------------------------------
//# Transformations

Transform create_transform()
      raises(DocException);

Transform acquire_internal_transform(in Canvas bias_canvas)
      raises(DocException);
```

```
void change_internal_transform(
      in Transform            transform,
      in Canvas               bias_canvas
)
      raises(DocException);

//#---------------------------------
//# Content Extent

void get_content_extent(out Point content_extent)
      raises(DocException);

void change_content_extent(in Point content_extent)
      raises(DocException);

//#---------------------------------
//# Drag & Drop

boolean is_droppable()
      raises(DocException);

void set_droppable(in boolean is_droppable)
      raises(DocException);

boolean is_dragging()
      raises(DocException);

void set_dragging(in boolean is_dragging)
      raises(DocException);

//#---------------------------------
//# Linking

void content_updated(in UpdateID change)
      raises(DocException);

void change_link_status(in LinkStatus status)
      raises(DocException);

LinkStatus get_link_status()
      raises(DocException);

boolean edit_in_link()
      raises(DocException);

//#---------------------------------
```

```
            //# Invalidation/Draw

            void invalidate(in Shape invalid_shape, in Canvas bias_canvas)
                  raises(DocException);

            void validate(in Shape valid_shape, in Canvas bias_canvas)
                  raises(DocException);

            void invalidate_active_border()
                  raises(DocException);

            void draw_active_border()
                  raises(DocException);

            //#---------------------------------
            //# Ref counting

            void close()
                  raises(DocException);
      };

};
```

Info

```
module CfDoc {

      interface Info :  Object
      {
            boolean show_part_frame_info(
                  in Facet                facet,
                  in boolean              allow_editing
            )
                  raises(DocException);
      };

};
```

MenuBar

```
module CfDoc {

      interface MenuBar :  RefCntObject
      {
            void display()
```

```
                raises(DocException);

            void add_menu_last(
                in MenuID               menu_id,
                in PlatformMenu         menu,
                in Part                 part
            )
                raises(DocException);

            void add_menu_before(
                in MenuID               menu_id,
                in PlatformMenu         menu,
                in Part                 part,
                in MenuID               before_id
            )
                raises(DocException);

            void remove_menu(in MenuID menu)
                raises(DocException);

            PlatformMenu get_menu(in MenuID menu)
                raises(DocException);

            boolean is_valid()
                raises(DocException);
        };

};
```

ObjectIterator

```
module CfDoc {

        interface ObjectIterator :  Object
        {
            void first(
                out ISOStr              key,
                out Object              object,
                out unsigned            long object_length
            )
                raises(DocException);

            void next(
                out ISOStr              key,
                out Object              object,
                out unsigned            long object_length
```

```
            )
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };

};
```

Shape

```
module CfDoc {

      interface Shape :  RefCntObject
      {
            //#--------------------------------
            //# Getters/Setters

            GeometryMode get_geometry_mode()
                  raises(DocException);

            void set_geometry_mode(in GeometryMode mode)
                  raises(DocException);

            void get_bounding_box(out Rect bounds)
                  raises(DocException);

            Shape set_rectangle(in Rect rect)
                  raises(DocException);

            void copy_polygon(out Polygon copy)
                  raises(DocException);

            Shape set_polygon(in Polygon polygon)
                  raises(DocException);

            PlatformShape get_platform_shape(
                  in GraphicsSystem       graphics_system
            )
                  raises(DocException);

            void set_platform_shape(
                  in GraphicsSystem       graphics_system,
                  in PlatformShape        platform_shape
            )
                  raises(DocException);
```

```
void reset()
      raises(DocException);

//#--------------------------------
//# Input/Output

void write_shape(in StorageUnit storage_unit)
      raises(DocException);

Shape read_shape(in StorageUnit storage_unit)
      raises(DocException);

//#--------------------------------
//# Comparison/Testing Functions

boolean is_same_as(in Shape compare_shape)
      raises(DocException);

boolean is_empty()
      raises(DocException);

boolean contains_point(in Point point)
      raises(DocException);

boolean is_rectangular()
      raises(DocException);

boolean has_geometry()
      raises(DocException);

//#--------------------------------
//# Geometry Operations

void copy_from(in Shape source_shape)
      raises(DocException);

Shape transform_by(in Transform transform)
      raises(DocException);

Shape inverse_transform(in Transform transform)
      raises(DocException);

Shape subtract(in Shape diff_shape)
      raises(DocException);
```

```
            Shape intersect(in Shape sect_shape)
                  raises(DocException);

            Shape union_with(in Shape union_shape)
                  raises(DocException);

            Shape outset(in Coordinate distance)
                  raises(DocException);
      };

};
```

Transform

```
module CfDoc {

      interface Transform :  RefCntObject
      {
            //#--------------------------------
            //# Getters

            TransformType get_type()
                  raises(DocException);

            void get_offset(out Point offset)
                  raises(DocException);

            void get_pre_scale_offset(out Point offset)
                  raises(DocException);

            void get_scale(out Point scale)
                  raises(DocException);

            void get_matrix(out Matrix matrix)
                  raises(DocException);

            boolean has_matrix()
                  raises(DocException);

            boolean is_same_as(in Transform compare_transform)
                  raises(DocException);

            //#--------------------------------
            //# Setters

            Transform reset()
```

```
      raises(DocException);

Transform set_matrix(in Matrix matrix)
      raises(DocException);

Transform copy_from(in Transform source_transform)
      raises(DocException);

//#---------------------------------
//# Geometric Operations

Transform set_offset(in Point point)
      raises(DocException);

Transform move_by(in Point point)
      raises(DocException);

Transform scale_by(in Point scale)
      raises(DocException);

Transform scale_down_by(in Point scale)
      raises(DocException);


Transform invert()
      raises(DocException);

Transform pre_compose(in Transform transform)
      raises(DocException);

Transform post_compose(in Transform transform)
      raises(DocException);

//#---------------------------------
//# Geometry Operations with Points & Shapes

void transform_point(inout Point point)
      raises(DocException);

void invert_point(inout Point point)
      raises(DocException);

void transform_shape(in Shape shape)
      raises(DocException);

void invert_shape(in Shape shape)
```

```
                raises(DocException);

            //#--------------------------------
            //# Input / Output

            void write_to(in StorageUnit storage_unit)
                raises(DocException);

            void read_from(in StorageUnit storage_unit)
                raises(DocException);
      };

};
```

Window

```
module CfDoc {

      interface Window :  RefCntObject
      {
            PlatformWindow get_platform_window()
                raises(DocException);

            Frame acquire_source_frame()
                raises(DocException);

            void set_source_frame(in Frame frame)
                raises(DocException);

            Frame get_root_frame()
                raises(DocException);

            Facet get_facet_under_point(in Point point)
                raises(DocException);

            boolean is_active()
                raises(DocException);

            void open()
                raises(DocException);

            void close()
                raises(DocException);

            void close_and_remove()
                raises(DocException);
```

```
            void show()
                  raises(DocException);

            void hide()
                  raises(DocException);

            boolean is_shown()
                  raises(DocException);

            boolean is_resizable()
                  raises(DocException);

            boolean is_root_window()
                  raises(DocException);

            boolean should_save()
                  raises(DocException);

            void set_should_save(in boolean should_save)
                  raises(DocException);

            boolean should_show_links()
                  raises(DocException);

            void set_should_show_links(in boolean should_show_links)
                  raises(DocException);

            void adjust_window_shape()
                  raises(DocException);

            ID get_id()
                  raises(DocException);

            boolean is_floating()
                  raises(DocException);

            Facet get_root_facet()
                  raises(DocException);
      };

};
```

WindowState

```
module CfDoc {
```

```
interface WindowState :  Object
{

      Window register_window(
            in PlatformWindow       new_window,
            in Type                 frame_type,
            in boolean              is_root_window,
            in boolean              is_resizable,
            in boolean              is_floating,
            in boolean              should_save,
            in Part                 root_part,
            in TypeToken            view_type,
            in TypeToken            presentation,
            in Frame                source_frame
      )
            raises(DocException);

      Window register_window_for_frame(
            in PlatformWindow       new_window,
            in Frame                frame,
            in boolean              is_root_window,
            in boolean              is_resizable,
            in boolean              is_floating,
            in boolean              should_save,
            in Frame                source_frame
      )
            raises(DocException);

      Window acquire_window(in ID id)
            raises(DocException);

      void internalize(in Draft draft)
            raises(DocException);

      void externalize(in Draft draft)
            raises(DocException);

      void set_default_window_titles(in Draft draft)
            raises(DocException);

      void open_windows(in Draft draft)
            raises(DocException);

      void close_windows(in Draft draft)
            raises(DocException);
```

```
            unsigned short get_window_count()
                  raises(DocException);

            unsigned short get_root_window_count(in Draft draft)
                  raises(DocException);

            unsigned short get_total_root_window_count()
                  raises(DocException);

            boolean is_Window(in PlatformWindow window)
                  raises(DocException);

            Window acquire_window_from_platform_window(in PlatformWindow
window)
                  raises(DocException);

            WindowIterator create_window_iterator()
                  raises(DocException);

            Window acquire_active_window()
                  raises(DocException);

            void set_base_menu_bar(in MenuBar the_menu_bar)
                  raises(DocException);

            MenuBar copy_base_menu_bar()
                  raises(DocException);

            void adjust_part_menus()
                  raises(DocException);

            MenuBar create_menu_bar(in PlatformMenuBar menu_bar)
                  raises(DocException);

            Canvas create_canvas(
                  in GraphicsSystem       graphics_system,
                  in PlatformCanvas       platform_canvas,
                  in boolean              is_dynamic,
                  in boolean              is_offscreen
            )
                  raises(DocException);

            Facet create_facet(
                  in Frame                frame,
                  in Shape                clip_shape,
```

```
                in Transform            external_transform,
                in Canvas               canvas,
                in Canvas               bias_canvas
            )
                raises(DocException);
        };
};
```

Appendix D - IDL for the Compound Core Facility

See “Appendix F - Types and Constants” for the data type and constant definitions.

Binding

```
module CfDoc {

      interface Binding :  Object
      {
            Editor choose_editor_for_part(
                  in StorageUnit          the_part_su,
                  in Type                 new_kind
            )
                  raises(DocException);

            ContainerSuite get_container_suite(
                  in ContainerType        container_type
            )
                  raises(DocException);
      };

};
```

Object

```
#include <LifeCycle.idl>
#include <Naming.idl>
#include <ObjectIdentity.idl>

module CfDoc {

      interface Object : CosLifeCycle::LifeCycleObject,
                  CosObjectIdentity::IdentifiableObject
      {
            void init_object()
                  raises(DocException);

            boolean has_extension(in Type extension_name)
                  raises(DocException);

            Extension acquire_extension(in Type extension_name)
                  raises(DocException);

            void release_extension(in Extension extension)
                  raises(DocException);
```

```
            Size purge(in Size size)
                  raises(DocException);
      };

};
```

Part

```
module CfDoc {

      interface Part : PersistentObject
      {
            //#================================
            //# Required Protocols

            //#--------------------------------
            //# Initialization

            void init_part(in StorageUnit storage_unit, in Part part_wrapper)
                  raises(DocException);

            void init_part_from_storage(
                  in StorageUnit          storage_unit,
                  in Part                 part_wrapper
            )
                  raises(DocException);

            //#--------------------------------
            //# From Layout Protocol

            void display_frame_added(in Frame frame)
                  raises(DocException);

            void display_frame_removed(in Frame frame)
                  raises(DocException);

            void display_frame_connected(in Frame frame)
                  raises(DocException);

            void display_frame_closed(in Frame frame)
                  raises(DocException);

            void attach_source_frame(in Frame frame, in Frame source_frame)
                  raises(DocException);

            void frame_shape_changed(in Frame frame)
```

```
      raises(DocException);

void view_type_changed(in Frame frame)
      raises(DocException);

void presentation_changed(in Frame frame)
      raises(DocException);

void sequence_changed(in Frame frame)
      raises(DocException);

void link_status_changed(in Frame frame)
      raises(DocException);

void containing_part_properties_updated(in Frame frame,
                              in StorageUnit property_unit)
      raises(DocException);

InfoType read_part_info(
      in Frame                frame,
      in StorageUnitView      storage_unit_view
      )
      raises(DocException);

void write_part_info(
      in InfoType             part_info,
      in StorageUnitView      storage_unit_view
)
      raises(DocException);

void clone_part_info(
      in DraftKey             key,
      in InfoType             part_info,
      in StorageUnitView      storage_unit_view,
      in Frame                scope
)
      raises(DocException);

ID open(in Frame frame)
      raises(DocException);

//#--------------------------------
//# From Imaging Protocol

void draw(in Facet facet, in Shape invalid_shape)
      raises(DocException);
```

```
void facet_added(in Facet facet)
      raises(DocException);

void facet_removed(in Facet facet)
      raises(DocException);

void geometry_changed(
      in Facet                facet,
      in boolean              clip_shape_changed,
      in boolean              external_transform_changed
)
      raises(DocException);

void highlight_changed(in Facet facet)
      raises(DocException);

void canvas_changed(in Facet facet)
      raises(DocException);

void canvas_updated(in Canvas canvas)
      raises(DocException);

unsigned long get_print_resolution(in Frame frame)
      raises(DocException);

//#--------------------------------
//# From Part Activation Protocol

boolean begin_relinquish_focus(
      in TypeToken            focus,
      in Frame                owner_frame,
      in Frame                proposed_frame
)
      raises(DocException);

void commit_relinquish_focus(
      in TypeToken            focus,
      in Frame                owner_frame,
      in Frame                proposed_frame
)
      raises(DocException);

void abort_relinquish_focus(
      in TypeToken            focus,
      in Frame                owner_frame,
```

```
      in Frame                proposed_frame
)
      raises(DocException);

void focus_acquired(in TypeToken focus, in Frame owner_frame)
      raises(DocException);

void focus_lost(in TypeToken focus, in Frame owner_frame)
      raises(DocException);


//#--------------------------------
//# From Part Persistance Protocol

void externalize_kinds(in TypeList kindset)
      raises(DocException);

void change_kind(in Type kind)
      raises(DocException);

//#--------------------------------
//# From UI Events Protocol

boolean handle_event(
      inout EventData         event,
      in Frame                frame,
      in Facet                facet,
      inout EventInfo         event_info
)
      raises(DocException);

void adjust_menus(in Frame frame)
      raises(DocException);

//#================================
//# Optional Protocols

//#--------------------------------
//# From Undo Protocol

void undo_action(in ActionData action_state)
      raises(DocException);

void redo_action(in ActionData action_state)
      raises(DocException);
```

```
void dispose_action_state(
      in ActionData           action_state,
      in DoneState            done_state
)
      raises(DocException);

void write_action_state(
      in ActionData           action_state,
      in StorageUnitView      storage_unit_view
)
      raises(DocException);

ActionData read_action_state(in StorageUnitView storage_unit_view)
      raises(DocException);

//#--------------------------------
//# From Drag-and-Drop Protocol

void fulfill_promise(in StorageUnitView promise_su_view)
      raises(DocException);

void drop_completed(in Part dest_part, in DropResult drop_result)
      raises(DocException);

DragResult drag_enter(
      in DragItemIterator     drag_info,
      in Facet                facet,
      in Point                where
)
      raises(DocException);

DragResult drag_within(
      in DragItemIterator     drag_info,
      in Facet                facet,
      in Point                where
)
      raises(DocException);

void drag_leave(in Facet facet, in Point where)
      raises(DocException);

DropResult drop(
      in DragItemIterator     drop_info,
      in Facet                facet,
      in Point                where
)
```

```
      raises(DocException);

//#--------------------------------
//# From Linking Protocol

LinkSource create_link(in ByteArray data)
      raises(DocException);

void reveal_link(in LinkSource link_source)
      raises(DocException);

void link_updated(in Link updated_link, in UpdateID change)
      raises(DocException);

// --- For Containing Parts ---

void embedded_frame_updated(in Frame frame, in UpdateID change)
      raises(DocException);

boolean edit_in_link_attempted(in Frame frame)
      raises(DocException);

//#--------------------------------
//# From Embedding Protocol

Frame request_embedded_frame(
      in Frame               containing_frame,
      in Frame               base_frame,
      in Shape               frame_shape,
      in Part                embed_part,
      in TypeToken           view_type,
      in TypeToken           presentation,
      in boolean             is_overlaid
)
      raises(DocException);

void remove_embedded_frame(in Frame embedded_frame)
      raises(DocException);

Shape request_frame_shape(
      in Frame               embedded_frame,
      in Shape               frame_shape
)
      raises(DocException);

void used_shape_changed(in Frame embedded_frame)
```

```
            raises(DocException);

        Shape adjust_border_shape(in Facet embedded_facet, in Shape shape)
            raises(DocException);

        StorageUnit acquire_containing_part_properties(in Frame frame)
            raises(DocException);

        boolean reveal_frame(
            in Frame                embedded_frame,
            in Shape                reveal_shape
        )
            raises(DocException);

        EmbeddedFramesIterator create_embedded_frames_iterator(
            in Frame                frame
        )
            raises(DocException);

        //#================================
        //# Do Not Override

        //#--------------------------------
        //# From Part Wrapping Protocol

        boolean is_real_part()
            raises(DocException);

        Part get_real_part()
            raises(DocException);

        void release_real_part()
            raises(DocException);
    };

};
```

PersistentObject

```
module CfDoc {

    interface PersistentObject :  RefCntObject
    {
        void init_persistent_object(in StorageUnit storage_unit)
            raises(DocException);
```

```
            void init_persistent_object_from_storage(in StorageUnit
storage_unit)
                  raises(DocException);

            void release_all()
                  raises(DocException);

            void externalize()
                  raises(DocException);

            StorageUnit get_storage_unit()
                  raises(DocException);

            ID get_id()
                  raises(DocException);

            void clone_into(
                  in DraftKey             key,
                  in StorageUnit          to_su,
                  in Frame                scope
            )
                  raises(DocException);
      };

};
```

RefCntObject

```
module CfDoc {

      interface RefCntObject :  Object
      {
            void init_ref_cnt_object()
                  raises(DocException);

            void acquire()
                  raises(DocException);

            void release()
                  raises(DocException);

            unsigned long get_ref_count()
                  raises(DocException);
      };

};
```

Session

```
module CfDoc {

      interface Session :  Object
      {
            void init_session()
                  raises(DocException);

            void get_user_name(out IText user_name)
                  raises(DocException);

            //#--------------------------------
            //# Global Getters/Setters

            Arbitrator get_arbitrator()
                  raises(DocException);

            void set_arbitrator(in Arbitrator arbitrator)
                  raises(DocException);


            Binding get_binding()
                  raises(DocException);

            void set_binding(in Binding binding)
                  raises(DocException);


            Dispatcher get_dispatcher()
                  raises(DocException);

            void set_dispatcher(in Dispatcher dispatcher)
                  raises(DocException);


            Clipboard get_clipboard()
                  raises(DocException);

            void set_clipboard(in Clipboard clipboard)
                  raises(DocException);


            DragAndDrop get_drag_and_drop()
                  raises(DocException);
```

```
void set_drag_and_drop(in DragAndDrop drag_and_drop)
      raises(DocException);


Info get_info()
      raises(DocException);

void  set_info(in Info info)
      raises(DocException);


LinkManager get_link_manager()
      raises(DocException);

void set_link_manager(in LinkManager link_manager)
      raises(DocException);


NameSpaceManager get_name_space_manager()
      raises(DocException);

void set_name_space_manager(
      in NameSpaceManager     name_space_manager
)
      raises(DocException);


StorageSystem get_storage_system()
      raises(DocException);

void set_storage_system(in StorageSystem storage_system)
      raises(DocException);


Translation get_translation()
      raises(DocException);

void set_translation(in Translation translation)
      raises(DocException);


Undo get_undo()
      raises(DocException);

void set_undo(in Undo undo)
```

```
                raises(DocException);


            WindowState get_window_state()
                raises(DocException);

            void set_window_state(in WindowState window_state)
                raises(DocException);

            //#---------------------------------
            //# Types and Token

            TypeToken tokenize(in Type type)
                raises(DocException);

            void remove_entry(in Type type)
                raises(DocException);

            boolean get_type(in TypeToken token, out Type type)
                raises(DocException);

            //#---------------------------------
            //# Data Interchange

            UpdateID unique_update_id()
                raises(DocException);
      };

};
```

Translation

```
module CfDoc {

      interface Translation :  Object
      {
            TranslateResult can_translate(in ValueType from_type)
                raises(DocException);

            TypeList get_translation_of(in ValueType from_type)
                raises(DocException);

            TranslateResult translate_view(
                in StorageUnitView      from_view,
                in StorageUnitView      to_view
            )
```

```
                raises(DocException);

            TranslateResult translate(
                in ValueType            from_type,
                in ByteArray            from_data,
                in ValueType            to_type,
                out ByteArray           to_data
            )
                raises(DocException);

            ValueType get_iso_type_from_platform_type(
                in PlatformType         platform_type,
                in PlatformTypeSpace    type_space
            )
                raises(DocException);

            PlatformType get_platform_type_from_iso_type(in ValueType type)
                raises(DocException);
      };

};
```

Undo

```
module CfDoc {

      interface Undo :  Object
      {
            void add_action_to_history(
                in Part                 which_part,
                in ActionData           action_data,
                in ActionType           action_type,
                in Name                 undo_action_label,
                in Name                 redo_action_label
            )
                raises(DocException);

            void undo_action()
                raises(DocException);

            void redo_action()
                raises(DocException);

            void mark_action_history()
                raises(DocException);
```

```
            void clear_action_history(in RespectMarksChoices respect_marks)
                  raises(DocException);

            void clear_redo_history()
                  raises(DocException);

            boolean peek_undo_history(
                  out Part                part,
                  out ActionData          action_data,
                  out ActionType          action_type,
                  out Name                action_label
            )
                  raises(DocException);

            boolean peek_redo_history(
                  out Part                part,
                  out ActionData          action_data,
                  out ActionType          action_type,
                  out Name                action_label
            )
                  raises(DocException);

            void abort_current_transaction()
                  raises(DocException);
      };
};
```

Appendix E - IDL for the Compound Utilities Facility

See “Appendix F - Types and Constants” for the data type and constant definitions.

DragItemIterator

```
module CfDoc {

      interface DragItemIterator : Object
      {
            StorageUnit first()
                  raises(DocException);

            StorageUnit next()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };

};
```

EmbeddedFramesIterator

```
module CfDoc {

      interface EmbeddedFramesIterator :  Object
      {
            void init_embedded_frames_iterator(in Part part)
                  raises(DocException);

            Frame first()
                  raises(DocException);

            Frame next()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);

            void part_removed()
                  raises(DocException);

            boolean is_valid()
                  raises(DocException);
```

```
            void check_valid()
                  raises(DocException);
      };

};
```

FocusOwnerIterator

```
module CfDoc {

      interface FocusOwnerIterator : Object
      {
            void init_focus_owner_iterator(
                  in TypeToken            focus,
                  in FocusModule          focus_module
            )
                  raises(DocException);

            Frame first()
                  raises(DocException);

            Frame next()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };

};
```

FocusSetIterator

```
module CfDoc {

      interface FocusSetIterator : Object
      {
            TypeToken first()
                  raises(DocException);

            TypeToken next()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };
```

```
};
```

FrameFacetIterator

```
module CfDoc {

      interface FrameFacetIterator : Object
      {
            Facet first()
                  raises(DocException);

            Facet next()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };

};
```

NameSpace

```
module CfDoc {

      interface NameSpace :  Object, CosNaming::NamingContext
      {
            ISOStr get_name()
                  raises(DocException);

            NSTypeSpec get_type()
                  raises(DocException);

            NameSpace get_parent()
                  raises(DocException);

            void set_type(in NSTypeSpec type)
                  raises(DocException);

            void unregister(in ISOStr key)
                  raises(DocException);

            boolean exists(in ISOStr key)
                  raises(DocException);
```

```
            void write_to_file(in ByteArray file)
                  raises(DocException);

            void read_from_file(in ByteArray file)
                  raises(DocException);

            void write_to_storage(in StorageUnitView view)
                  raises(DocException);

            void read_from_storage(in StorageUnitView view)
                  raises(DocException);
      };

};
```

NameSpaceManager

```
module CfDoc {

      interface NameSpaceManager :  Object
      {
            NameSpace create_name_space(
                  in ISOStr               space_name,
                  in NameSpace            inherits_from,
                  in unsigned             long num_expected_entries,
                  in NSTypeSpec           type
            )
                  raises(DocException);

            void delete_name_space(in NameSpace the_name_space)
                  raises(DocException);

            NameSpace has_name_space(in ISOStr space_name)
                  raises(DocException);
      };

};
```

ObjectNameSpace

```
module CfDoc {

      interface ObjectNameSpace :  NameSpace
      {
            void register(in ISOStr key, in Object object)
```

```
                raises(DocException);

            boolean get_entry(in ISOStr key, out Object object)
                raises(DocException);

            ObjectIterator create_iterator()
                raises(DocException);
      };

};
```

PlatformTypeList

```
module CfDoc {

      interface PlatformTypeList :  Object
      {
            void add_last(in PlatformType type)
                raises(DocException);

            void remove(in PlatformType type)
                raises(DocException);

            boolean contains(in PlatformType type)
                raises(DocException);

            unsigned long count()
                raises(DocException);

            PlatformTypeListIterator create_platform_type_list_iterator()
                raises(DocException);
      };

};
```

PlatformTypeListIterator

```
module CfDoc {

      interface PlatformTypeListIterator :  Object
      {
            boolean is_not_complete()
                raises(DocException);

            PlatformType first()
```

```
                raises(DocException);

            PlatformType next()
                raises(DocException);
      };

};
```

StorageUnitRefIterator

```
module CfDoc {

      interface StorageUnitRefIterator : Object
      {
            void first(out StorageUnitRef ref)
                raises(DocException);

            void next(out StorageUnitRef ref)
                raises(DocException);

            boolean is_not_complete()
                raises(DocException);
      };

};
```

TypeList

```
module CfDoc {

      interface TypeList : Object
      {
            void add_last(in Type type)
                raises(DocException);

            void remove(in Type type)
                raises(DocException);

            boolean contains(in Type type)
                raises(DocException);

            unsigned long count()
                raises(DocException);

            TypeListIterator create_type_list_iterator()
```

```
                  raises(DocException);
      };

};
```

TypeListIterator

```
module CfDoc {

      interface TypeListIterator : Object
      {
            boolean is_not_complete()
                  raises(DocException);

            Type first()
                  raises(DocException);

            Type next()
                  raises(DocException);
      };

};
```

ValueIterator

```
module CfDoc {

      interface ValueIterator :  Object
      {
            void first(out ISOStr key, out ByteArray value)
                  raises(DocException);

            void next(out ISOStr key, out ByteArray value)
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };

};
```

ValueNameSpace

```
module CfDoc {
```

```
      interface ValueNameSpace :  NameSpace
      {
            void register(in ISOStr key, in ByteArray value)
                  raises(DocException);

            boolean get_entry(in ISOStr key, out ByteArray value)
                  raises(DocException);

            ValueIterator create_iterator()
                  raises(DocException);
      };

};
```

WindowIterator

```
module CfDoc {

      interface WindowIterator : Object
      {
            Window first()
                  raises(DocException);

            Window next()
                  raises(DocException);

            Window last()
                  raises(DocException);

            Window previous()
                  raises(DocException);

            boolean is_not_complete()
                  raises(DocException);
      };
};
```

Appendix F - Types and Constants

```
//# Primitive and Platform-Independent Types

module CfDoc {

//#============================================================================
//# Scalar Types
//#============================================================================

//#----------------------------------------------------------------------------
//# Common Types
//#----------------------------------------------------------------------------
typedef string ISOStr;                  // 7 bit ascii. No embedded NULLs. NULL
                                        //  terminated.

typedef ISOStr Type;                    // Used for storage types, focus types,
                                        //  data types, etc.

typedef unsigned long TypeToken;        // Tokenized form of Type

typedef unsigned long ID;               // Storage Unit IDs

typedef unsigned long PersistentObjectID;// Persistent Object ID used for scripting

typedef Type ValueType;                 // Used to identify the type of the value,
                                        // e.g., TEXT, PICT and so on. Obviously, the type
                                        // doesn't need to be an OSType.

typedef long Error;                     // >= 32-bit exception code

typedef unsigned long Token;            // >= 32-bit unsigned value for tokens

typedef unsigned long Size;             // >= 32-bit unsigned value for size

typedef unsigned long InfoType;         // The type for the part info stored in a frame

typedef unsigned long Flags;            // >= 32-bit unsigned value for flags

typedef sequence<octet> ByteArray;      // ByteArray should be used for foreign types
                                        // larger than 4 bytes and for variable length data
                                        // in general

typedef ByteArray ContainerID;
typedef ISOStr ContainerType;

typedef Type ObjectType;

typedef Type FocusType;                 // This is a string, which can be tokenized

typedef ByteArray ActionData;           // Action data for undo/redo

enum FramePosition
{
           frame_behind,
           frame_in_front
};

enum TraversalType
```

```
{
            top_down,
            bottom_up,
            children_only
};

enum SiblingOrder
{
            front_to_back,
            back_to_front
};

enum NSTypeSpec
{
            ns_data_type_object,
            ns_data_type_value
};

enum RespectMarksChoices
{
            dont_respect_marks,
            respect_marks
};

enum ActionType
{
            single_action,
            begin_action,
            end_action
};

//#--------------------------------------------------------------------------
//# Imaging Types
//#--------------------------------------------------------------------------
typedef long Fixed;                     // 16.16 fixed-point value
typedef long Fract;                     //  2.30 fixed-point value

struct Matrix       // Transform matrix for translation, scaling,
{                                       // skewing, rotation or any combination.
            Fixed m[3][3];
};

typedef short GraphicsSystem;    // Type of graphics system

typedef short TransformType;

enum GeometryMode
{
            lose_geometry,              // Toss out geometric info to save time or space.
            preserve_geometry,          // Preserve geometric info as long as possible
            needs_geometry              // Must keep geometric info.
};

enum LinkStatus
{
            in_link_destination,
            in_link_source,
            not_in_link
};
```

```
enum Highlight
{
            no_highlight,
            full_highlight,
            dim_highlight
};

//#--------------------------------------------------------------------------
//# For DragAndDrop
//#--------------------------------------------------------------------------
enum DropResult
{
            drop_fail,
            drop_copy,
            drop_move,
            drop_unfinished              // Used only when DragAndDrop::StartDrag
                                         // returns immediately.
};

//#--------------------------------------------------------------------------
//# For Link
//#--------------------------------------------------------------------------
typedef unsigned long LinkKey;

//#--------------------------------------------------------------------------
//# For Translation
//#--------------------------------------------------------------------------
enum TranslateResult
{
            cannot_translate,
            can_translate
};

typedef unsigned long PlatformTypeSpace;

//#--------------------------------------------------------------------------
//# For StorageUnit
//#--------------------------------------------------------------------------
typedef ID ValueIndex;
typedef ID StorageUnitID;

const unsigned long storage_unit_ref_size = 4;
typedef octet StorageUnitRef[storage_unit_ref_size];

typedef unsigned long StorageUnitKey;

typedef ISOStr PropertyName;
typedef ISOStr StorageUnitName;
typedef unsigned long PositionCode;

//#--------------------------------------------------------------------------
//# For Document
//#--------------------------------------------------------------------------
typedef ID DocumentID;

//#--------------------------------------------------------------------------
//# For Draft
//#--------------------------------------------------------------------------
```

```
enum PurgePriority
{
            invisible_blocks,
            all_blocks,
            visible_blocks
};

enum CloneKind
{
            clone_copy,
            clone_cut,
            clone_paste,
            clone_drop_copy,
            clone_drop_move,
            clone_to_link,
            clone_from_link
};

enum DraftPermissions
{
            dp_none,
            dp_transient,
            dp_read_only,
            dp_shared_write,
            dp_exclusive_write
};

typedef    ID DraftID;
typedef    ISOStr DraftName;

//#-----------------------------------------------------------------------------
//# for Undo protocol
//#-----------------------------------------------------------------------------
enum DoneState
{
            done,
            undone,
            redone
};

//#-----------------------------------------------------------------------------
//# Exceptions
//#-----------------------------------------------------------------------------
exception DocException
{
            Error error;
};


};

module CfDoc {

//#=============================================================================
//# General errors including Object
//#=============================================================================

const Error min_error                   =-29849;// range of err codes
const Error min_used_error              =-29755;
```

```
const Error max_error                    =-29600;


const Error no_error                     = 0;


const Error err_undefined                = -29600;

const Error err_already_notified         =-29601;

const Error err_illegal_null_input       =-29602;

const Error err_illegal_null_dispatch_module_input=-29603;
const Error err_illegal_null_facet_input =-29604;
const Error err_illegal_null_frame_input =-29605;
const Error err_illegal_null_part_input          =-29606;
const Error err_illegal_null_transform_input=-29607;
const Error err_illegal_null_storage_system_input=-29608;
const Error err_illegal_null_token_input =-29609;
const Error err_illegal_null_shape_input =-29610;
const Error err_illegal_null_storage_unit_input=-29611;
const Error err_illegal_null_property_input      =-29612;
const Error err_illegal_null_su_cursor_input=-29613;
const Error err_illegal_null_container_input    =-29614;
const Error err_illegal_null_document_input=-29615;
const Error err_illegal_null_draft_input =-29616;
const Error err_illegal_null_value_type_input       =-29617;
const Error err_illegal_null_id_input    =-29618;

const Error err_value_out_of_range       =-29619;

const Error err_insufficient_info_in_params=-29620;

const Error err_object_not_initialized   =   -29621;

const Error err_out_of_memory            = -29622;

const Error err_not_implemented          = -29623;

const Error err_invalid_persistent_format=-29624;

const Error err_sub_class_responsibility = -29625;

//#-------------------------------------------------------------------------------
//# Object (& subclasses)
//#-------------------------------------------------------------------------------

const Error err_unsupported_extension         =-29626;
const Error err_invalid_extension           =-29627;
const Error err_unknown_extension                =-29628;
const Error err_invalid_object_type      =-29629;
const Error err_invalid_persistent_object_id=-29630;
const Error err_invalid_persistent_object=-29631;

//#-------------------------------------------------------------------------------
//# RefCountObject (& subclasses)
//#-------------------------------------------------------------------------------

const Error err_zero_ref_count           = -29632;
```

```
const Error err_ref_count_greater_than_zero       = -29633;

const Error err_ref_count_not_equal_one          = -29634;

//#--------------------------------------------------------------------------
//# Iterators
//#--------------------------------------------------------------------------

const Error err_iterator_out_of_sync       = -29635;
const Error err_iterator_not_initialized       = -29636;
const Error err_invalid_iterator                = -29755;

//#==========================================================================
//# Parts return these errors
//#==========================================================================

const Error err_cannot_embed             =-29637;
const Error err_does_not_undo            =-29638;
const Error err_no_promises              =-29639;
const Error err_does_not_drop            =-29640;
const Error err_does_not_link            =-29641;
const Error err_part_not_wrapper          =-29642;

//#==========================================================================
//# Core
//#==========================================================================

//#--------------------------------------------------------------------------
//# NameSpaceManager, NameSpace
//#--------------------------------------------------------------------------

const Error err_key_already_exists        = -29643;

const Error err_invalid_ns_name          =-29644;

const Error err_invalid_ns_type          =-29645;

//#--------------------------------------------------------------------------
//# PartWrapper
//#--------------------------------------------------------------------------

const Error err_part_in_use              = -29646;

//#--------------------------------------------------------------------------
//# PartWrapper
//#--------------------------------------------------------------------------

const Error err_invalid_itext_format     = -29647;

//#==========================================================================
//# Imaging
//#==========================================================================

const Error err_invalid_graphics_system    = -29648;

//#--------------------------------------------------------------------------
//# Shape
//#--------------------------------------------------------------------------
```

```
const Error err_no_shape_geometry        =-29649; //# shape has no polygon representation

//#---------------------------------------------------------------------------
//# Transform
//#---------------------------------------------------------------------------

const Error err_transform_err            =-29650; //# illegal op on xform

const Error err_invalid_platform_shape   =-29651;

//#===========================================================================
//# Layout
//#===========================================================================

//#---------------------------------------------------------------------------
//# Facet
//#---------------------------------------------------------------------------

const Error err_canvas_not_found         =-29652;

const Error err_unsupported_frame_position_code=-29653;

const Error err_invalid_facet            =-29654;

const Error err_facet_not_found          =-29655;
const Error err_canvas_has_no_owner          =-29656;

//#---------------------------------------------------------------------------
//# Frame
//#---------------------------------------------------------------------------

const Error err_not_root_frame           =-29657;
const Error err_illegal_recursive_embedding=-29658;

const Error err_invalid_frame            =-29659;

const Error err_frame_has_facets          =-29660;

const Error err_invalid_link_status          =-29754;

//#===========================================================================
//# Memory
//#===========================================================================

const Error err_invalid_block            =-29661;    //# bad heap or damaged object

//#===========================================================================
//# Storage
//#===========================================================================

const Error err_unsupported_pos_code        =-29664;
const Error err_invalid_permissions      =-29665;

const Error err_cannot_create_container   =-29666;
const Error err_cannot_open_container      =-29667;

const Error err_container_does_not_exist =-29668;
```

```
const Error err_document_does_not_exist  =-29669;

const Error err_draft_does_not_exist     =-29670;
const Error err_draft_has_been_deleted   =-29671;

const Error err_invalid_storage_unit     =-29672;

const Error err_illegal_operation_on_su  =-29673;

const Error err_su_value_does_not_exist  =-29674;

const Error err_illegal_non_topmost_draft=-29675;

//#------------------------------------------------------------------------------
//# Storage unit
//#------------------------------------------------------------------------------

const Error err_no_value_at_that_index      =-29676;
const Error err_cannot_add_property      =-29677;

const Error err_unfocused_storage_unit      =-29678;
const Error err_invalid_storage_unit_ref    =-29679;
const Error err_storage_unit_locked      =-29680;
const Error err_invalid_storage_unit_key    =-29681;
const Error err_storage_unit_not_locked      =-29682;
const Error err_invalid_draft_key        =-29683;
const Error err_cloning_in_progress      =   -29684;
const Error err_value_index_out_of_range =   -29685;

const Error err_invalid_value_type       =-29686;

const Error err_illegal_property_name    =-29687;
const Error err_property_does_not_exist  =-29688;

//#------------------------------------------------------------------------------
//# Draft
//#------------------------------------------------------------------------------

const Error err_no_draft_properties      = -29689;
const Error err_cannot_create_frame      =-29690;
const Error err_cannot_acquire_frame       =-29691;
const Error err_cannot_create_part        =-29692;
const Error err_cannot_acquire_part       =-29693;
const Error err_cannot_create_link        =-29694;
const Error err_cannot_acquire_link       =-29695;

const Error err_invalid_id               =-29696;

const Error err_inconsistent_clone_kind        =-29697;
const Error err_invalid_clone_kind        =-29698;
const Error err_invalid_destination_draft= -29699;
const Error err_move_into_self           =-29700;
const Error err_null_destination_frame   =-29701;

//#------------------------------------------------------------------------------
//# Document
//#------------------------------------------------------------------------------

const Error err_invalid_below_draft      =-29702;
```

```
const Error err_cannot_collapse_drafts      =-29704;
const Error err_non_empty_draft          =-29705;
const Error err_no_previous_draft        =-29706;
const Error err_outstanding_draft        =-29707;
const Error err_invalid_draft_id         =-29708;
const Error err_cannot_change_permissions    =-29709;

//#------------------------------------------------------------------------
//# Storage system
//#------------------------------------------------------------------------

const Error err_container_exists         =-29710;

//#------------------------------------------------------------------------
//# LinkSpec
//#------------------------------------------------------------------------

const Error err_cannot_get_external_link    =-29712;
const Error err_no_link_spec_value        =-29713;
const Error err_unknown_link_spec_version    =-29714;
const Error err_corrupt_link_spec_value      =-29715;

//#------------------------------------------------------------------------
//# Link
//#------------------------------------------------------------------------

const Error err_not_exported_link        =-29716;
const Error err_broken_link              =-29717;
const Error err_cannot_reveal_link       =-29718;
const Error err_corrupt_link             =-29719;
const Error err_link_already_exported    =-29720;
const Error err_no_link_content          =-29721;
const Error err_cannot_register_dependent=-29722;

//#------------------------------------------------------------------------
//# LinkSource
//#------------------------------------------------------------------------

const Error err_not_imported_link        =-29723;
const Error err_invalid_link_key         =-29724;
const Error err_broken_link_source          =-29725;
const Error err_corrupt_link_source      =-29726;
const Error err_cannot_find_link_source_edition=-29727;// cannot locate source of cross
                                         // document link because edition file does not exist
const Error err_cannot_find_link_source  =-29728;// cannot locate source of cross document
                                         // link
const Error err_already_imported_link    =-29729;// cannot create link due to internal error
const Error err_unknown_update_id        =-29730;
const Error err_cannot_establish_link    =-29731;

//#------------------------------------------------------------------------
//# LinkManager
//#------------------------------------------------------------------------

const Error err_no_edition_manager       =-29732;
const Error err_doc_not_saved            =-29733;

//#------------------------------------------------------------------------
//# Data Interchange Dialogs
```

```
//#------------------------------------------------------------------------------

const Error err_null_facet_input         =-29734;
const Error err_null_link_info_input     =-29735;
const Error err_null_link_info_result_input=-29736;
const Error err_null_paste_as_result_input=-29737;

//#------------------------------------------------------------------------------
//# DragAndDrop
//#------------------------------------------------------------------------------

const Error err_no_drag_manager          =-29738;
const Error err_no_drag_system_storage      =-29739;
const Error err_drag_item_not_found        =-29740;
const Error err_cannot_allocate_drag_item    =-29741;
const Error err_unknown_drag_image_type      =-29742;
const Error err_drag_tracking_exception    =-29743;

//#------------------------------------------------------------------------------
//# Clipboard
//#------------------------------------------------------------------------------

const Error err_background_clipboard_clear    =-29744;
const Error err_illegal_clipboard_clone_kind=-29745;

//#==============================================================================
//# UI
//#==============================================================================

//#------------------------------------------------------------------------------
//# Arbitrator
//#------------------------------------------------------------------------------

const Error err_focus_already_registered    =-29746;
const Error err_focus_not_registered      =-29747;

//#------------------------------------------------------------------------------
//# Undo
//#------------------------------------------------------------------------------

const Error err_cannot_mark_action        =-29749;
const Error err_empty_stack              =-29750;
const Error err_no_begin_action          =-29751;
const Error err_cannot_add_action        =-29752;

//#------------------------------------------------------------------------------
//# WindowState
//#------------------------------------------------------------------------------

const Error err_cannot_create_window      =-29753;

};

module CfDoc {


//#------------------------------------------------------------------------------
//# Prefixes
//#------------------------------------------------------------------------------
```

```
const ISOStr iso_prefix = “+//ISO 9070/ANSI::113722::US::CI LABS::”;

//#--------------------------------------------------------------------------
//# Miscellaneous
//#--------------------------------------------------------------------------
const unsigned long no_wait = 0;         // For the wait parameter to Lock()

//#--------------------------------------------------------------------------
//# Tokens
//#--------------------------------------------------------------------------
const TypeToken null_type_token = 0;

//#--------------------------------------------------------------------------
//# IDs
//#--------------------------------------------------------------------------
const ID null_id= 0;
const ID id_all                          = 0;
const ID index_all= 0;
const ID id_wild= 0;

//#--------------------------------------------------------------------------
//# Object Types
//#--------------------------------------------------------------------------
const ObjectType part_object             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ObjectType:Part”;
const ObjectType frame_object            = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ObjectType:Frame”;
const ObjectType non_persistent_frame_object= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ObjectType:NonPersistentFrame”;

//#--------------------------------------------------------------------------
//# NameSpaces
//#--------------------------------------------------------------------------
const ISOStr container_suite             =“+//ISO 9070/ANSI::113722::US::CI
LABS::ContainerSuite”;
const ISOStr kind                        =“+//ISO 9070/ANSI::113722::US::CI
LABS::KindCategories”;
const ISOStr editor_kinds                =“+//ISO 9070/ANSI::113722::US::CI
LABS::EditorKinds”;
const ISOStr viewer                      =“+//ISO 9070/ANSI::113722::US::CI LABS::Viewers”;
const ISOStr kind_old_mac_os_type        =“+//ISO 9070/ANSI::113722::US::CI
LABS::KindOldMacOSType”;
const ISOStr editor_platform_kind        =“+//ISO 9070/ANSI::113722::US::CI
LABS::EditorPlatformKind”;
const ISOStr editor_help_file            =“+//ISO 9070/ANSI::113722::US::CI
LABS::EditorHelpFile”;

const ISOStr sys_pref_container_suites=  “+//ISO 9070/ANSI::113722::US::CI
LABS::SysPrefContainerSuites”;
const ISOStr sys_pref_editor_kinds       =“+//ISO 9070/ANSI::113722::US::CI
LABS::SysPrefEditorKinds”;
const ISOStr sys_pref_editor_categories= “+//ISO 9070/ANSI::113722::US::CI
LABS::SysPrefEditorCategories”;

const ISOStr editor_user_string          =“+//ISO 9070/ANSI::113722::US::CI
LABS::EditorUserString”;
const ISOStr kind_user_string            =“+//ISO 9070/ANSI::113722::US::CI
LABS::KindUserString”;
```

```
const ISOStr category_user_string        =“+//ISO 9070/ANSI::113722::US::CI
LABS::CategoryUserString”;

const OSType name_mappings               =‘nmap’;
const ISOStr simple_viewer               =“”;

//#-------------------------------------------------------------------------
//# Editor IDs
//#-------------------------------------------------------------------------
const Editor no_editor = 0;
const ISOStr black_box_handler_of_last_resort = “Apple::NoPart”;

//#-------------------------------------------------------------------------
//# Categories
//#-------------------------------------------------------------------------
const ISOStr category_plain_text         =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Text:Plain”;
const ISOStr category_styled_text        =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Text:Styled”;
const ISOStr category_drawing            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Drawing”;
const ISOStr category_3d_graphic         =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:3DGraphic”;
const ISOStr category_painting           =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Painting”;
const ISOStr category_movie              =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Movie”;
const ISOStr category_sampled_sound      =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:SampledSound”;
const ISOStr category_structured_sound=  “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:StructuredSound”;
const ISOStr category_chart              =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Chart”;
const ISOStr category_formula            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Formula”;
const ISOStr category_spreadsheet        =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Spreadsheet”;
const ISOStr category_table              =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Table”;
const ISOStr category_database           =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Database”;
const ISOStr category_query              =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Query”;
const ISOStr category_connection         =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Connection”;
const ISOStr category_script             =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Script”;
const ISOStr category_outline            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Outline”;
const ISOStr category_page_layout        =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:PageLayout”;
const ISOStr category_presentation       =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Presentation”;
const ISOStr category_calendar           =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Calendar”;
const ISOStr category_form               =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Form”;
const ISOStr category_executable         =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Executable”;
```

```
const ISOStr category_compressed         =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Compressed”;
const ISOStr category_control_panel      =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:ControlPanel”;
const ISOStr category_control            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Control”;
const ISOStr category_personal_info      =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:PersonalInfo”;
const ISOStr category_space              =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Space”;
const ISOStr category_project            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Project”;
const ISOStr category_signature          =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Signature”;
const ISOStr category_key                =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Key”;
const ISOStr category_utility            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Utility”;
const ISOStr category_mailing_label      =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:MailingLabel”;
const ISOStr category_locator            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Locator”;
const ISOStr category_printer            =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Printer”;
const ISOStr category_time               =“+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Category:Time”;

//#-------------------------------------------------------------------------
//# PlatformTypeSpace
//#-------------------------------------------------------------------------
const unsigned long platform_file_type=  1;
const unsigned long platform_data_type=  2;

//#-------------------------------------------------------------------------
//# International Text Format Constants
//#-------------------------------------------------------------------------
const short iso_10646_1993_base_encoding =0;    // UCS

//#-------------------------------------------------------------------------
//# View Type Sizes
//#-------------------------------------------------------------------------
const short tiny_icon_size               =12;// 12x12 pixels
const short small_icon_size              =16;// 16x16 pixels
const short large_icon_size              =32;// 32x32 pixels
const short thumbnail_size               =64;// 64x64 pixels

//#-------------------------------------------------------------------------
//# Imaging
//#-------------------------------------------------------------------------
const GraphicsSystem no_graphics_system = 0;
           // Graphics systems are of course platform dependent, but their numeric IDs
           // should be globally registered to avoid overlaps; otherwise confusion may
           // occur when documents are moved between platforms, or objects on different
           // platforms attempt to communicate.

const TransformType identity_xform         = 0;// Identity (no-op) transform
const TransformType translate_xform        = 1;// Pure translation/offset
const TransformType scale_xform             = 2;// Pure scale, no offset
const TransformType scale_translate_xform = 3;// Scale and offset
```

```
const TransformType linear_xform           = 4;// Scale/rotate/skew, but no offset
const TransformType linear_translate_xform= 5;// Linear as above, but with offset
const TransformType perspective_xform  = 6;// Perspective: m[0][2]!=0 or m[1][2]!=0

const TransformType unknown_xform          =-1;// Type not known yet [internal use only]
const TransformType invalid_xform          = 7;// Bad matrix [internal use only]

//#------------------------------------------------------------------------
//# ViewTypes and Presentations
//#------------------------------------------------------------------------
const Type view_as_small_Icon= "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ViewAs:SmallIcon";
const Type view_as_large_Icon= "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ViewAs:LargeIcon";
const Type view_as_thumbnail= "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ViewAs:Thumbnail";
const Type view_as_frame                = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:ViewAs:Frame";

const Type pres_default                 = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Presentation:Default";

//#------------------------------------------------------------------------
//# Drag And Drop
//#------------------------------------------------------------------------
const unsigned long drag_is_in_source_frame= 1; //0x00000001
const unsigned long drag_is_in_source_part= 2; //0x00000002

const unsigned long drop_is_in_source_frame= drag_is_in_source_frame;
const unsigned long drop_is_in_source_part= drag_is_in_source_part;
const unsigned long drop_is_move         = 4; //0x00000004
const unsigned long drop_is_copy         = 8; //0x00000008
const unsigned long drop_is_paste_as     = 16; //0x00000010

//#------------------------------------------------------------------------
//# Interchange
//#------------------------------------------------------------------------
const Type drag_image_region_handle= "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:DragImage:RegionHandle";
const ValueType hfs_promise             = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:HFSPromise";
const ValueTypetype_all                 = 0;

const UpdateID unknown_update = 0;

const LinkUpdateResult link_update_suspend = 0;
const LinkUpdateResult link_update_stop = 1;
const LinkUpdateResult link_update_continue = 2;

const PasteAsMergeSetting paste_as_merge = 1;
const PasteAsMergeSetting paste_as_embed = 3;
const PasteAsMergeSetting paste_as_merge_only = 0;
const PasteAsMergeSetting paste_as_embed_only = 2;

//#------------------------------------------------------------------------
//# For StorageUnit
//#------------------------------------------------------------------------
const StorageUnitKey null_key = 0;
```

```
const unsigned long pos_undefined= 4294967295; //0xFFFFFFFF
const unsigned long pos_same             = 0;
const unsigned long pos_all              = 1;
const unsigned long pos_first_sib= 2;
const unsigned long pos_last_sib= 3;
const unsigned long pos_next_sib= 4;
const unsigned long pos_prev_sib= 5;
const unsigned long pos_first_below= 6;
const unsigned long pos_last_below= 7;
const unsigned long pos_first_above= 8;
const unsigned long pos_last_above= 9;
const unsigned long pos_reserved11= 10;
const unsigned long pos_reserved12= 11;
const unsigned long pos_reserved13= 12;
const unsigned long pos_reserved14= 13;
const unsigned long pos_reserved15= 14;
const unsigned long pos_mwrap            = 16; //0x10
const unsigned long pos_top              = 32; //0x20

//#--------------------------------------------------------------------------
//# For Document
//#--------------------------------------------------------------------------
const DocumentID default_document = 1;

const ContainerType default_file_container= “Apple:ContainerType:File”;
const ContainerType default_memory_container= “Apple:ContainerType:Memory”;

const ContainerType bento_embedded_container= “Apple:Bento:ContainerType:Embedded”;
const ContainerType bento_file_container = “Apple:Bento:ContainerType:File”;
const ContainerType bento_memory_container= “Apple:Bento:ContainerType:Memory”;

//#--------------------------------------------------------------------------
//# Events
//#--------------------------------------------------------------------------
const short evt_null                     = 0;
const short evt_mouse_down               = 1;
const short evt_mouse_up                 = 2;
const short evt_key_down                 = 3;
const short evt_key_up                   = 4;
const short evt_auto_key                 = 5;
const short evt_update                   = 6;
const short evt_disk                     = 7;
const short evt_activate                 = 8;
const short evt_os                       = 15;
const short evt_bg_mouse_down            = 16;// mouse-down while in background [special]

const short evt_menu                     = 98;
const short evt_mouse_down_embedded      = 99;
          // a mouse-down in an embedded frame, or the active border
                                         // sent to the container
const short evt_mouse_up_embedded        = 100;
          // a mouse-up in an embedded frame, or the active border
                                         // sent to the container
const short evt_mouse_down_border        = 101;
          // a mouse-down in an embedded frame, or the active border
                                         // sent to the container
const short evt_mouse_up_border          = 102;/
          / a mouse-up in an embedded frame, or the active border
                                         // sent to the container
```

```
const short evt_window                    = 103;
                                          // used to offer events in title bar to root part
const short evt_mouse_enter               = 104;
const short evt_mouse_within              = 105;
const short evt_mouse_leave               = 106;
const short evt_bg_mouse_down_embedded    = 107;
                                          // a mouse-down in an embedded frame, when process
is inactive
                                          // sent to the container
const short evt_exit                       = 108;
           // Dispatched immediately before session is destructed.

const short md_in_desk                    = 0;
const short md_in_menu_bar                = 1;
const short md_in_sys_window              = 2;
const short md_in_content                 = 3;
const short md_in_drag                    = 4;
const short md_in_grow                    = 5;
const short md_in_go_away                 = 6;
const short md_in_zoom_in                 = 7;
const short md_in_zoom_out                = 8;

//#-------------------------------------------------------------------------
//# Shell
//#-------------------------------------------------------------------------
const OSType shell_signature = 'odtm';


};

module CfDoc {

//=========================================================================
// Constants
//=========================================================================

//# Prefixes

const PropertyName prop_pre_metadata     = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:";
const PropertyName prop_pre_annotation   = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Annotation:";

//# Persistent object

const PropertyName prop_name             = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Name";
const PropertyName prop_comments         = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Comments";
const PropertyName prop_create_date      = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:CreationDate";
const PropertyName prop_mod_date         = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ModifiedDate";
const PropertyName prop_mod_user         = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ModifiedUser";
const PropertyName prop_object_type      = "+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ObjectType";
```

```
const PropertyName prop_storage_unit_type= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:StorageUnitType”;

//# Part

const PropertyName prop_contents         = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:Contents”;
const PropertyName prop_is_stationery    = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:IsStationery”;
const PropertyName prop_preferred_editor = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:PreferredEditor”;
const PropertyName prop_preferred_kind   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:PreferredKind”;
const PropertyName prop_display_frames     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Annotation:DisplayFrames”;
const PropertyName prop_custom_icon      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:CustomIcon”;

//# Frame

const PropertyName prop_containing_frame = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ContainingFrame”;
const PropertyName prop_graphics_system  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:GraphicsSystem”;
const PropertyName prop_frame_shape      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:FrameShape”;
const PropertyName prop_internal_transform= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:InternalTransform”;
const PropertyName prop_bias_transform   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:BiasTransform”;
const PropertyName prop_part             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Part”;
const PropertyName prop_part_info        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:PartInfo”;
const PropertyName prop_view_type        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ViewType”;
const PropertyName prop_presentation     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Presentation”;
const PropertyName prop_frame_group      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:FrameGroup”;
const PropertyName prop_sequence_number  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:SequenceNumber”;
const PropertyName prop_is_root          = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:IsRoot”;
const PropertyName prop_is_subframe      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:IsSubFrame”;
const PropertyName prop_is_overlaid      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:IsOverlaid”;
const PropertyName prop_is_frozen        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:IsFrozen”;
const PropertyName prop_does_propagate_events= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:DoesPropagateEvents”;
const PropertyName prop_link_status      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:LinkStatus”;

const PropertyName prop_window_properties= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowProperties”;

//# Link
```

```
const PropertyName prop_link             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Link”;
const PropertyName prop_link_source      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:LinkSource”;
const PropertyName prop_link_spec        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:LinkSpec”;
const PropertyName prop_source_part      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:SourcePart”;
const PropertyName prop_edition_alias    = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:EditionAlias”;
const PropertyName prop_link_section     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:LinkSection”;
const PropertyName prop_link_content_su  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:LinkContentSU”;
const PropertyName prop_auto_update      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:AutoUpdate”;
const PropertyName prop_update_id        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:UpdateID”;
const PropertyName prop_change_limit     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ChangeLimit”;
const PropertyName prop_change_time      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ChangeTime”;
const PropertyName prop_content_kinds_used= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ContentKindsUsed”;
const PropertyName prop_original_id      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:OriginalID”;
const PropertyName prop_original_draft   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:OriginalDraft”;
const PropertyName prop_original_clone_kind= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:OriginalCloneKind”;
const PropertyName prop_reserved_section_ids= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ReservedSectionIDs”;

//# Window

const PropertyName prop_window           = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Window”;
const PropertyName prop_window_rect       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowRect”;
const PropertyName prop_window_title       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowTitle”;
const PropertyName prop_window_proc_id     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowProcID”;
const PropertyName prop_window_is_visible    = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowVisible”;
const PropertyName prop_window_has_close_box  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowHasCloseBox”;
const PropertyName prop_window_has_zoom_box   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowHasZoomBox”;
const PropertyName prop_window_is_resizable  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowIsResizable”;
const PropertyName prop_window_is_root_window = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowIsRootWindow”;
const PropertyName prop_window_is_floating   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowIsFloating”;
const PropertyName prop_window_ref_con     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:WindowRefCon”;
```

```
const PropertyName prop_root_frame       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:RootFrame”;
const PropertyName prop_source_frame       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:SourceFrame”;
const PropertyName prop_should_show_links    = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:ShouldShowLinks”;

//# Draft
const PropertyName prop_root_part_su     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:Draft:RootPartStorageUnit”;
          //# Draft History
const PropertyName prop_draft_number     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Draft:Number”;
const PropertyName prop_draft_comment      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Draft:Comment”;
          //# Link Manager
const PropertyName prop_edition_id       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Draft:EditionID”;
const PropertyName prop_section_id       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:Draft:SectionID”;
          //# WindowState
const PropertyName prop_root_frame_list     = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:RootFrameList”;
const PropertyName prop_draft_saved_date = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Metadata:DraftSavedDate”;

//# Data Interchange

const PropertyName prop_proxy_contents   = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:ProxyContents”;
const PropertyName prop_content_frame    = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:ContentFrame”;
const PropertyName prop_suggested_frame_shape= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:SuggestedFrameShape”;
const PropertyName prop_clone_kind_used  = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:CloneKindUsed”;

//# Drag & Drop

const PropertyName prop_mouse_down_offset= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:MouseDownOffset”;

//# Printing
const PropertyName prop_page_setup       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Property:PageSetup”;

};

module CfDoc {

//============================================================================
// Constants
//============================================================================

module CfDoc {

//# Universal Types
```

```
const ValueType  type_boolean           = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Boolean”;
const ValueType  type_ushort            = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:UShort”;
const ValueType  type_sshort            = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:SShort”;
const ValueType  type_ulong             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:ULong”;
const ValueType  type_slong             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:SLong”;

const ValueType  type_iso_str           = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:ISOStr”;
const ValueType  type_iso_str_list      = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:ISOStrList”;

const ValueType  type_intl_text         = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:IntlText”;

const ValueType  type_time_t            = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Time_T”;

const ValueType  type_point             = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Point”;
const ValueType  type_rect              = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Rect”;
const ValueType  type_polygon           = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Polygon”;

const ValueType  type_icon_family       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:IconFamily”;

const ValueType  type_transform         = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Transform”;

const ValueType  type_editor            = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:Editor”;

const Type       type_strong_storage_unit_ref= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:StorageUnitRef”;
const Type       type_weak_storage_unit_ref= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:StorageUnitRef”;

const Type       type_strong_storage_unit_refs= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:StorageUnitRefs”;
const Type       type_weak_storage_unit_refs= “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:StorageUnitRefs”;

const ValueType  type_link_spec         = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:LinkSpec”;
const ValueType  type_clone_kind        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:CloneKind”;

const ValueType  type_object_type       = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:ObjectType”;

};
```

```
const ValueType  type_link_spec          = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:LinkSpec”;
const ValueType  type_clone_kind         = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:CloneKind”;

const ValueType  type_object_type        = “+//ISO 9070/ANSI::113722::US::CI
LABS::OpenDoc:Type:ObjectType”;

};
```

Appendix G - Glossary

abstract class A class used only to derive other classes. An abstract class is never instantiated. Compare **concrete class**.

action data Information stored in the undo object's action history that allows a part to reverse the effects of an undoable action.

action history The cumulative set of reversible actions available at any one time, maintained by the undo object.

action subhistory A subset of action data added to the undo object's action history by a part in a modal state. The part can then remove the subhistory from the action history without affecting earlier actions.

action types constants that define whether an undoable action is a single-stage action (such as a cut) or part of a two-stage action (such as a drag-move).

activate To have received the selection focus. A frame activates itself when a mouse-down event occurs within it. On most platforms, a window is activated when it is to the front, or when the cursor passes over it.

active frame The frame that has the selection focus. Editing takes place in the active frame; it displays the selection or insertion point. The active frame almost always has the keystroke focus also.

active part The part displayed in the active frame. The active part controls the part-specific palettes and menus, and its content contains the selection or insertion point. The active part can be displayed in one or more frames, only one of which is the active frame.

active shape A shape that describes the portion of a facet within which a part expects to receive user events. If, for example, an embedded part's used shape and active shape are identical, the containing part both draws and accepts events in the unused areas within the embedded part's frame.

ancestor See **superclass.**

application A software product that allows a user to create and manipulate documents. See also **application component, conventional application.**

application component An OpenDoc component that the user employs to create, edit, or view document parts. Part editors and part viewers are application components; they replace the functionality of **conventional applications.**

arbitrator An OpenDoc object that manages negotiation among parts about ownership of **shared resources.** Examples of such resources are the menu focus, the selection focus, the keystroke focus, and the serial ports.

auxiliary storage unit Compare main storage unit.

base class See **superclass.**

Bento A document storage architecture, built on top of a platform's native file system, that allows for the creation, storage, and retrieval of compound documents. The OpenDoc storage system on some platforms is based on Bento.

bias transform A transform that is applied to measurements in a part's coordinate system to change them into **platform-normal coordinates.**

binding The process of selecting an executable code module based on type information.

border See **frame border.**

bundled frame A frame whose contents do not respond to user events. A mouse click within a bundled frame selects the frame, but does not activate the frame.

canvas The platform-specific drawing environment on which frames are laid out. Each window or printing device has one drawing canvas. See also **static canvas** and **dynamic canvas.**

category See **part category.**

change ID (1) A number used to identify a particular instance of Clipboard contents. (2) A number used to identify a particular instance of link source data.

child class See **subclass.**

circular link A configuration of links in which changes to a link's destination directly or indirectly affect its source.

class A programming entity comprising data structures and methods, from which objects that are instances of the class are created.

class hierarchy The structure by which classes are related through **inheritance.**

Clipboard A system-maintained buffer that provides a facility for transferring data within and across documents.

Clipboard focus Access to the Clipboard. The part with the Clipboard focus can read from and write to the Clipboard.

clip shape A shape that defines the limits of drawing within a facet.

clone To copy an object and all its referenced objects. When you clone an object, that object plus all other objects to which there is a **strong persistent reference** in the cloned object are copied.

close For a frame, to remove it from memory but not from storage. A closed frame is not permanently removed from its document. Compare **remove.**

Common Object Request Broker Architecture (CORBA) A standard promulgated by the Object Management Group industry consortium for defining interactions among objects.

component A software product that functions in the OpenDoc environment. Part editors, part viewers, and services are examples of components. See also **application component, service component.**

compound document A single document containing multiple heterogeneous data types, each presented and edited by its own software. A compound document is made up of **parts.**

concrete class A class designed to be instantiated. Compare **abstract class.**

container (1) A holder of persistent data (documents), part of an OpenDoc **container suite.** (2) See **container part, container application.**

container application An application program that has been modified to support embedding of OpenDoc parts. A container application functions as both document shell and part editor for the root part.

container suite A set of OpenDoc classes that implement persistent storage. The container suite consists of containers, documents, drafts, and storage units.

container part A part that is capable of embedding other parts or links to other parts. Compare **noncontainer part.**

containing frame The display frame of a containing part. Each embedded frame has one containing frame; each containing frame can have one or more embedded frames.

containing part The part in which a frame is embedded. Each embedded frame has one containing part; each containing part has one or more embedded frames.

containment A relationship between objects wherein an object of one class contains a reference to an object of another class. Compare **inheritance.**

content See **part content.**

content area The potentially visible area of a part as viewed in a frame or window. If the content area is greater than the area of the frame or window, only a portion of the part can be viewed at a time.

content element A user-visible data item presented by a part's content model. Content elements can be manipulated through the graphical or scripting interface to a part.

content extent The vertical dimension of the content area of a part in a frame. Content extent is used to calculate **bias transforms.**

content model The specification of a part's contents (the data types of its content elements) and its content operations (the actions that can be performed on it and the interactions among its content elements).

content object A content element that can be represented as an object and thus accessed and manipulated through semantic events.

content operation A user action that manipulates a content element.

content storage unit The main storage unit of the Clipboard, drag-and-drop object, link source object, or link object.

content transform The composite transform that converts from a part's content coordinates to its canvas coordinates.

conventional application An application that directly handles events and opens documents, and is wholly responsible for manipulating, storing, and retrieving all of the data in its documents. Compare **application component.**

coordinate bias The difference between a given coordinate system and **platform- normal coordinates.** Coordinate bias typically involves both a change in axis polarity and an offset.

current draft A specially designated draft that is the most recent draft of a document.

current frame During drawing, the frame that is being drawn or within which editing is occurring.

customizable Characteristic of a scriptable part that also defines content objects and operations for interface elements such as menus and buttons.

derived class See **subclass.**

descendant See **subclass.**

destination content The content at the destination of a link. It is a copy of the **source content.**

destination part A part that displays, through a link, information that resides in another part. Compare **source part.**

dispatcher The OpenDoc object that directs user events and semantic events to the correct part.

dispatch module An OpenDoc object used by the dispatcher to dispatch events of a certain type to part editors.

display frame A frame in which a part is displayed. A part's display frames are created by and embedded in its containing part. Compare **embedded frame.**

document In OpenDoc, a user-organized collection of parts, all stored together.

document part See **part**.

document process A thread of execution that runs the document shell program. The document process provides the interface between the operating system and part editors: it accepts events from the operating system, provides the address space into which parts are read, and provides access to the window system and other features.

document shell A program that provides an environment for all the parts in a document. The shell maintains the major document global databases: storage, window state, arbitrator, and dispatcher. This code also provides basic document behavior like document creation, open, save, print, and close. OpenDoc provides a default document shell for each platform.

document window A window that displays an OpenDoc document. The edges of the content area of the window represent the frame border of the document's root part. The OpenDoc document shell manages opening and closing document windows. Compare **part window.**

draft A configuration of a document, defined at a certain point in time by the user. A document is made up of a set of drafts.

drag and drop A facility of OpenDoc that allows users to apply direct manipulation to move or copy data.

drag-copy A drag-and-drop operation in which the dragged data remains at the source, and a copy is inserted at the destination.

drag-move A drag-and-drop operation in which the dragged data is deleted from the source and inserted at the destination.

drawing canvas See **canvas.**

dynamic canvas A drawing canvas that can potentially be changed, such as a window that can be scrolled or paged to display different portions of a part's data. Compare **static canvas.**

edit-in-place See **in-place editing.**

editor See **part editor.**

editor of last resort The part editor that displays any part for which there is no available part editor on the system. The editor of last resort typically displays a gray rectangle representing the part's frame.

editor preferences A dialog box, accessed through the Edit menu, in which the user can view and change preferences for the part editor of the currently active part.

embed To place one part within another so that, although its data is stored with the containing part's data and its frame is contained within the containing part's frame, it retains its identity as a separate part. Compare **incorporate.**

embedded content Content displayed in an embedded frame. A part editor does not directly manipulate embedded content. Compare **intrinsic content.**

embedded frame A frame that is part of a containing part's content, and within which an embedded part is displayed. The embedded frame itself is considered intrinsic content of the

containing part; the part displayed within the frame is considered embedded content of the containing part.

embedded-frames list A containing part's list of all the frames embedded within it.

embedded part A part that is embedded in another part. The data for an embedded part is stored within the same draft as its containing part. An embedded part is copied during a duplication of its containing part. An embedded part may itself be a containing part, unless it is a **noncontainer part.** Compare **linked part.**

exception An execution error or abnormal condition detected by the runtime facilities of the system.

exclusive focus A focus that can be owned by only one frame at a time. The selection focus, for example, is exclusive; the user can edit within only one frame at a time. Compare **non-exclusive focus.**

externalize See **write.**

external transform A transform that is applied to a facet to position, scale, or otherwise transform the facet and the image drawn within it. The external transform locates the facet in the coordinate space of its frame's containing part. Compare **internal transform.**

extracted draft A draft that is extracted from a document into a new document.

facet An object that describes where a frame is displayed on a canvas.

factory method A method in one class that creates an instance of another class.

fidelity The faithfulness of translation attained (or attainable) between data of different part kinds. For a given part kind, other part kinds are ranked in fidelity by the level at which their editors can translate its data without loss.

focus A designation of a shared resource such as menus, selection, keystrokes, and serial ports. The part that owns a focus has use of that shared resource.

focus set A group of foci requested as a unit.

frame A bounded portion of the content area of a part, defining the location of an embedded part. The edge of a frame marks the boundary between intrinsic content and embedded content. A frame can be a rectangle or any other, even irregular, shape.

frame border A visual indication of the boundary of a frame. The appearance of the frame border indicates the state of the frame (active, inactive, or selected). The frame border is drawn and manipulated by the containing part or by OpenDoc, not by the part within the frame.

frame coordinate space The coordinate space in which a part's frame shape, used shape, active shape, and clip shape are defined.

frame group A set of its display frames that a part designates as related, for purposes such as flowing content from one frame to another. Each frame group has its own **group ID;** frames within a frame group have a **frame sequence.**

frame negotiation The process of adjusting the size and shape of an embedded frame. Embedded parts can request changes to their frames, but the containing parts control the changes that are granted.

frame sequence The order of frames in a frame group.

frame shape A shape that defines a frame and its border, expressed in terms of the frame's local coordinate space.

frame transform The composite transform that converts from a part's frame coordinates to its canvas coordinates

graphics system A specific drawing architecture. Some graphics systems (such as Display PostScript) are available on more than one platform; some platforms support more than one graphics system).

group ID A number that identifies a frame group, assigned by the group's containing part.

icon A small, type-specific picture that represents a part. Possible iconic view types for displaying a part include as a (standard) icon, small icon, or thumbnail; the other possible view type is in a **frame.**

identity transform A transform that has no effect on points to which it is applied.

inclusions list A list of part kinds that can be embedded in a given part. A part can define and use its own inclusions list in order to restrict embedding into itself.

incorporate To merge the data from one part into the contents of another part so that the merged data retains no separate identity as a part. Compare **embed.**

inheritance A relationship between classes wherein one class (the subclass) shares the type and methods of another class (the superclass).

in-place editing Manipulation by a user of data in an embedded part without leaving the context of the document in which the part is displayed—without, for example, opening a new window for the part.

inside-out activation A mode of user interaction in which a mouse click anywhere in a document activates the smallest possible enclosing frame and performs the appropriate selection action on the content element at the click location. OpenDoc uses inside-out selection. Compare **outside-in activation.**

instance See **object.**

instantiate To cause an object of a class to be created in memory at runtime.

Interface Definition Language (IDL) A syntax created by IBM to describe the interface of classes that can be compiled by the SOM compiler.

internalize See **read.**

internal transform A transform that positions, scales, or otherwise transforms the image of a part drawn within a frame. Compare **external transform.**

interoperability Access to an OpenDoc part or document from different platforms or with different software systems.

intrinsic content The content elements native to a particular part, as opposed to the content of parts embedded within it. Compare **embedded content.**

invalidate To mark an area of a canvas (or facet, or frame) as in need of redrawing.

invalid shape The area of a frame, facet, or canvas that needs redrawing. Update events cause redrawing of the invalid area.

invariant An aspect of the internal state of an object that must be maintained for the object to behave properly according to its design.

ISO string A null-terminated 7-bit ASCII string.

keystroke focus The destination of keystroke events. The part whose frame has the keystroke focus handles keystroke events. See also **selection focus.**

keystroke focus frame The frame to which keystroke events are to be sent.

kind See **part kind.**

layout The process of arranging frames and content elements in a document for drawing.

link (1) A persistent reference to a part or to a set of content elements of a part. (2) An OpenDoc object that represents a link destination.

link destination The portion of a part's content area that represents the destination of a link.

link source The portion of a part's content area that represents the source of a link.

link specification An object, placed on the Clipboard or in a drag-and-drop object, from which the source part (the part that placed the data) can construct a link if necessary.

link status The link-related state (in a link source, in a link destination, or not in a link) of a frame.

lock To acquire exclusive access to. A part must lock a link source object or link object before accessing its data.

main storage unit The storage unit that holds the contents property (`prop_contents`) of a part. A part's main storage unit, plus possibly other storage units referenced from it, holds all of a part's content.

member function See **method.**

method An operation that manipulates the data of a particular class of objects.

modal focus The right to display modal dialog boxes. A part displaying a modal dialog must first obtain the modal focus, so that other parts cannot do the same until the first part is finished.

monitor A special use of a dispatch module, in which it is installed in order to be notified of events, but does not dispatch them.

monolithic application See **conventional application.**

name space An object consisting of a set of text strings used to identify kinds of objects or classes of behavior, for code-binding purposes. For

example, OpenDoc uses name spaces to identify part kinds and categories, and to identify object extensions.

name-space manager An OpenDoc object that creates and deletes **name spaces.**

noncontainer part A part that cannot itself contain embedded parts. Compare **container part.**

nonexclusive focus A focus that can be owned by more than one frame at a time. OpenDoc supports the use of nonexclusive foci. Compare **exclusive focus.**

object A programming entity, existing in memory at run time, that is an individual specimen of a particular **class.**

object specifier A designation of a content object within a part, used to determine the target of a semantic event. Object specifiers can be names ("blue rectangle") or logical designations ("word 1 of line 2 of embedded frame 3").

OLE 2.0 Object Linking and Embedding, Microsoft Corporation's compound documentarchitecture.

outside-in activation A mode of user interaction in which a mouse click anywhere in a document activates the largest possible enclosing frame that is not already active. Compare **inside-out activation.**

overlaid frame An embedded frame that floats above the content (including other embedded frames) of its containing part, and thus need not engage in frame negotiation with the containing part.

override To replace a method belonging to a superclass with a method of the same name in a subclass, in order to modify its behavior.

owner For a canvas, the part that created the canvas and attached it to a facet. The owner is responsible for transferring the results of drawing on the canvas to its parent canvas.

parent canvas The canvas closest above a canvas in the facet hierarchy. If there is a single off screen canvas attached to an embedded facet in a window, for example, the window canvas (attached to the root facet) is the parent of the off screen canvas.

parent class See **superclass.**

part A portion of a compound document; it consists of document content, plus—at runtime—a part editor that manipulates that content. The content is data of a given structure or type, such as text, graphics, or video; the code is a part editor. In programming terms, a part is an object, an instantiation of a subclass of the class `Part`. To a user, a part is a single set of information displayed and manipulated in one or more frames or windows. Same as **document part.**

part category A general classification of the format of data handled by a part editor. Categories are broad classes of data format, meaningful to end-users, such as "text", "graphics" or "table". Compare **part kind**.

part container See **container part.**

part content The portion of a part that describes its data. in programming terms, the part content is represented by the instance variables of the part object; it is the state of the part, and is the portion of it that is stored persistently. To the user, there is no distinction between part and part content; the user considers both the part content alone, and the content plus its part editor, as a part. See also **intrinsic content, embedded content.** Compare **part editor; part.**

part editor An application component that can display and change the data of a part. It is the executable code that provides the behavior for the part. Compare **part content, part viewer.**

part ID An identifier that uniquely names a part within the context of a document. This ID represents a storage unit ID within a particular draft of a document.

part info (1) Part-specific data, of any type or size, used by a part editor to identify what should be displayed in a particular frame or facet and how it should be displayed. (2) User-visible information about a given part, displayed in a dialog box accessed through a menu command.

part kind A specific classification of the format of data handled by a part editor. A kind specifies the specific data format handled by, and possibly native to, a part editor. Kinds are meaningful to end-users, and have designations such as such as "MacWrite 2.0" or "QuickTime 1.0". Compare **part category**.

part property One of a set of user-accessible characteristics of a part or its frame. The user can modify some part properties, such as the name of

a part; the user cannot modify some other part properties, such as part category. Each part property is stored as a distinct **property** in the storage unit of the part or its frame.

part table A list of all the parts contained within a document, plus associated data.

part viewer A part editor that can display and print, but not change, the data of a part. Compare **part editor.**

part window A window that displays an embedded part by itself, for easier viewing or editing. Any part that is embedded in another part can be opened up into its own part window. The part window is separate from, and has a slightly different appearance than, the **document window** displaying the entire document the part is embedded within.

persistence The quality of an entity such as a part, link, or object, that allows it to span separate document launches and transport to different computers. For example, a part written to persistent storage is typically written to a hard disk.

persistent reference A number, stored somewhere within a storage unit, that refers to another storage unit in the same document. Persistent references permit complex runtime object relationships to be stored externally, and later reconstructed.

platform A hardware/software operating environment. For example, OpenDoc is being implemented on the Macintosh, Windows, and OS/2 platforms.

platform-normal coordinates The native coordinate system for a particular platform. OpenDoc performs all layout and drawing in platform-normal coordinates; to convert from another coordinate system to platform-normal coordinates requires application of a **bias transform.**

position code A parameter (to a storage unit's Focus method) with which you specify the desired property or value to access.

presentation A particular style of display for a part—for example, outline or expanded for text, or wire-frame or solid for graphic objects. A part can have multiple presentations, each with its own rendering, layout, and user-interface behavior. See also **view type.**

promise A specification of data to be transferred at a future time. If a data transfer involves a very large amount of data, the source part can choose to put out a promise instead of actually writing the data to a storage unit.

property In the OpenDoc storage system, a component of a storage unit. A property defines a kind of information (such as "name" or "contents") and contains one or more data streams, called **values,** that consist of information of that kind. Properties in a stored part are accessible without the assistance of a part editor. See also **part property.**

protocol The programming interface through which a specific task or set of related tasks is performed. The drag-and-drop protocol, for example, is the set of calls that a part editor makes (and responds to) in order to support the dragging of items into or out of its content.

proxy content data, associated with a single embedded frame written to the Clipboard (or drag-and-drop object or link-source object), that the frame's original containing part wanted associated with the frame, such as a drop shadow or other visual adornment. Proxy content is absent if intrinsic content as well as an embedded frame was written.

purge To free non-critical memory, usually by writing or releasing cached data. In low-memory situations, OpenDoc can ask a part editor or other objects to purge memory.

read For a part or other OpenDoc object, to transform its persistent form in a storage unit into an appropriate in-memory representation, which can be a representation of the complete object or only a subset of it, depending on the current display requirements for the object. Same as **internalize**; compare **write.**

reference A pointer to (or other representation of) an object, used to gain access to the object when needed.

reference count The number of references to an object. Objects that are reference-counted, such as windows and parts, cannot be deleted from memory unless their reference counts are zero.

release To delete a reference to an object. For a reference-counted object, releasing it decrements its reference count.

remove For a frame, to permanently delete it from its document, as well as from memory. Compare **close.**

revert To return a draft to the state it had just after its last save.

root facet The facet that displays the root frame in a document window.

root frame The frame in which the root part of a document is displayed. The root frame shape is the same as the content area of the document window.

root part The part that forms the base of a document and establishes its basic editing, embedding, and printing behavior. A document has only one root part, which can contain content elements and perhaps other, embedded parts. Any part can be a root part.

root window See **document window.**

save To write all the data of all parts of a document (draft) to persistent storage.

select To designate as the locus of subsequent editing operations. If the user selects an embedded part, that part's frame border takes on an appearance that designates it as selected. The embedded part itself is not activated at this stage.

selection focus The location of editing activity. The part whose frame has the selection focus is the active part, and has the selection or insertion point. See also **keystroke focus.**

service or **part service** An OpenDoc component that provides services to document parts, instead of creating or viewing them. Compare **part editor.**

shape A description of a geometric area of a drawing canvas**.**

shared resource A facility used by multiple parts. Examples of shared resources are the menu focus, selection focus, keystroke focus, and serial ports. See also **arbitrator.**

sibling A frame or facet at the same level of embedding as another frame or facet within the same containing frame or facet. Sibling frames and facets are **z-ordered** to allow for overlapping.

source content The content at the source of a link. It is copied into the link and thence into the **destination content.**

source frame (1) An embedded frame whose part that has been opened up into its own **part window.** (2) The frame to which other **synchronized frames** are attached.

source part A part that contains information that is displayed in another part through a link. Compare **destination part.**

split-frame view A display technique for windows or frames, in which two or more facets of a frame display different scrolled portions of a part's content.

static canvas A drawing canvas that cannot be changed once it has been rendered, such as a printer page. Compare **dynamic canvas.**

stationery A part that opens by copying itself and opening the copy into a window, leaving the original stationery part unchanged.

storage system The OpenDoc mechanism for providing persistent storage for documents and parts. The storage system object must provide unique identifiers for parts as well as cross-document links. It stores parts as a set of standard properties plus type-specific content data.

storage unit In the OpenDoc storage system, an object that represents the basic unit of persistent storage. Each storage unit has a list of properties, and each property contains one or more data streams called values.

storage-unit cursor A prefocused storage unit/property/value designation, created to allow swift focusing on frequently accessed data.

storage unit ID A unique identifier of a storage unit within a draft.

strong persistent reference A persistent reference that, when the storage unit containing the reference is cloned, causes the referenced storage unit to be copied also. Compare **weak persistent reference**.

subclass A class derived from another class (its **superclass**), from which it inherits type and behavior. Also called derived class or descendant.

subframe A frame that is both an embedded frame in, and a display frame of, a part. A part can create an embedded frame, make it a subframe of its own display frame, and then display itself in that subframe.

subsystem A broad subdivision of the interface and capabilities of OpenDoc, divided along shared-library boundaries. The OpenDoc subsystems include shell, storage, layout, imaging, user events, and semantic events. Individual OpenDoc subsystems are replaceable.

superclass A class from which another class (its **subclass**) is derived. Also called ancestor, base class, or parent class. See also **inheritance.**

synchronized frames Separate frames that display the same representation of the same part, and should therefore be updated together. In general, if an embedded part has two or more editable display frames of the same presentation, those frames (and all their embedded frames) should be synchronized.

thumbnail A large (64-by-64 pixels) icon used to represent a part. The icon is typically a miniature representation of the layout of the part content.

token A short, codified representation of a string. The session object creates tokens for ISO strings.

transform A geometric transformation that can be applied to a graphic object when it is rendered, such as moving, scaling, or rotation. Different platforms and different graphics systems have transforms with different capabilities.

translation The conversion of one type of data to another type of data. Specifically, the conversion of data of one part kind to data of another part kind. The translation object is an OpenDoc wrapper for platform- specific translation capabilities. Note that translation can involve loss of **fidelity.**

translator A software utility, independent of OpenDoc, that converts data from one format to another. A translator may, for example, convert text in the format used by one word processor into a format readable by a different one. The translation capability of OpenDoc relies on the availability of translators.

undo To rescind a command, negating its results. The Undo object is an object that holds command history information in order to support the Undo capability of OpenDoc.

used shape A shape that describes the portion of a frame that a part actually uses for drawing; that is, the part of the frame that the containing part should not draw over.

user event A message, sent to a part by the dispatcher, that pertains only to the state of the part's graphical user interface, not directly to its contents. User events include mouse clicks and keystrokes, and they deliver information about, among other things, window locations and scroll bar positions.

user-interface part A part without content elements, representing a unit of a document's user interface. Buttons and dialog boxes, for example, can be user-interface parts.

validate To mark a portion of a canvas (or facet, or frame) as no longer in need of redrawing. Compare **invalidate.**

value In the OpenDoc storage system, a data stream associated with a property in a storage unit. Each property has a set of values, and there can be only one value of a given data type for each property.

viewer See **part viewer.**

view type The basic visual representation of a part. Supported view types include frame, icon, small icon, and thumbnail.

weak persistent reference A persistent reference that, when the storage unit containing the reference is cloned, is ignored; the referenced storage unit is not copied. Compare **strong persistent reference.**

window An area of a computer display in which information is presented to users in a graphic user interface, typically containing one or more content areas and controls, such as scroll bars, enabling the user to manipulate the display. Window systems are platform-specific.

window canvas The canvas attached to the root facet of a window. Every window has a window canvas.

window-content transform The composite transform that converts from a part's content coordinates to its window coordinates.

window-frame transform The composite transform that converts from a part's frame coordinates to its window coordinates.

window state An object that lists the set of windows that are open at a given time. Part editors can alter the window state, and the window state can be persistently stored.

write For a part or other OpenDoc object, to transform its in-memory representation into a persistent form in a storage unit. Same as **externalize**; compare **read.**

z-ordering The front-to-back ordering of sibling frames used to determine clipping and event handling when frames overlap.

Index