```
document = [session newDocument];

section = [document sectionAt:0];

[[section styleSheet] setDefaultStyle:
    [[styleSheet designModel]
    styleNamed: "Report Right"
    type: "Columns"]];
```

At this point the loan report could place Titles to introduce the report, and perhaps a Table to hold columnar data. For the sake of simplicity, we instead insert multiple lines into the existing story.

```
while(loadRecord = [mccaDB nextRecord])
{
    [story insertString:
        [loanRecord loanAmount]];
    [story insertString: "\n"];
}

[document saveAs: "loadReport"];
```

Again, we could vary the report format by adding text Emphasis styles, placing Rules to separate loan activity on a weekly basis, placing Heads for category of activity by loan officer, adding Headers and Footers to display report name, page number, date printed, or footnoting exceptional account activity.

In short, the API reflects the majority of features available through the Pages interface in an object/message format. The segregation of design and content allow the developer to interface simply with Pages, easily integrate data and manipulate style.

Appendix-B, Examples

This section to demonstrates how the Pages API might actually be used in an MCCA setting. The intent is to outline the general interaction with Pages and to suggest implementations. These are by no means the only ways to use the API. Also, any example code may or may not reflect the actual calls, though do reflect the general approach to using the API.[1]

Example 1: Loan Application

The MCCA objective is to gather information about an applicant, verify credit history, compute payment schedule and premiums, and produce a finished contract which the applicant signs.

The MCCA provider creates the database schemas, credit telecommunication links and processing, loan algorithms, and custom application package. Pages provides a collection of customizable classes which are created, assembled and printed to produce the finished loan application.

Once a customer information record has been entered and/or retrieved, the loan application establishes a connection to pages by instantiating an API-managing object (e.g. [PagesClient new]). This object is responsible for communicating instructions with the Pages formatting engine across a Distributed Object connection. The extent and type of this communication may be altered by subclassing.

After hooking into Pages, the loan application creates a document object from a pre-defined custom template. This template has includes key "fields" delimited by special characters (e.g. «applicant-name»), which are updated using search and replace by appropriate record content.

```
session = [PagesClient new];

...

document = [session
    newDocumentFromTemplate:[session
    templateNamed:"LoanTemplate"]];

story = [[document sectionAt:0] story];

[story replaceAll:"«applicant-name»"
    withString: [loanRecord applicantName]];

[story replaceAll:"«loan-amount»"
    withString:[loanRecord loanAmount]];

...

[document print];

[document close];
```

As the loan officer quits the loan application, the API controlling object is freed which terminates the connection to Pages.

Example 2: Loan Activity

The MCCA objective is to summarize loan activity during a given period, and save the result to a file.

The MCCA provider has created the application as described above, including the Pages Remote Object library. The load application is launched, and the API-managing object is instantiated. The default Design Model is changed to "Victory", a new document is created, and the column-style changed to "Report Right". Finally the report is saved.

```
session = [PagesClient new];

...

[session newDocUsesDesign:[session
    designNamed:"Victory"]];
```

[1] Beta documentation includes actual code examples.

Initials Operations

1. Get/Set Initial Style
2. Place Initial in Story Paragraph
3. Delete Initial

Initials are specialized treatment of a single text character at the beginning of a paragraph.

Special Paragraph Operations

1. Get/Set SpecialParagraph Style
2. Place Special Paragraph
3. Place Special Paragraph with Selection
4. Place Special Paragraph with String
5. Move Special Paragraph
6. Delete Special Paragraph

The Special Paragraph Elements allows visually distinct treatment of individual paragraphs within a BodyText Story. Support for creating custom Special Paragraphs is not provided in the PRO API. Suitable substitute are to use document Templates, StyleSets, or a custom Design Model.

Tab Operations

1. WIP

The Tab Element is a new Pages Element for release 1.1. API specification is forthcoming.

Element Operations

1. Place Element
2. Delete Element
3. Move Element
4. Set/Get Element Style
5. Get Page for Element (current page for floating elements, first page for multi-page elements)

These are operations shared by (nearly) all elements.

Other Operations and Notes

1. List Elements of Type
2. Find Element with String

These operations are provided to navigate between pieces and/or to query the document.

The inclusion of operations which are inherently UI based (such as Element tracking feedback), and not easily translated into appropriate API will be postponed until a future release of the API.

A Border Element, like a Caption, must be associated with another Element, and not all Elements will accept a Border. Failure to attach on create indicates Element rejection.

Table Operations

1 Get/Set Table Style
2 Get/Set Style Floating/Fixed
3 Get/Set Element Span (In-Column, Stub Column, Full Width)
4 Place Table
5 Move Table
6 Delete Table
7 Select Cell Range
8 Justify Cell Range
9 Apply Cell Borders to Range
10 Add Table Row
11 Delete Table Row
12 Show/Hide Table Title
13 Show/Hide Table Label
14 Show/Hide Table Summary
15 Get Story at Cell

The Table Element is a framed collection of text cells with rows and columns. Pages Tables have 3 distinct cell areas, two of which may be optionally visible. In addition, a Table title area contains text in a typeface similar to headline elements. Each Table cell contains a Story, which accepts emphasis and embellishment treatment like BodyText. In-Column Tables may be either fixed or floating.

Listing Operations

1 Get/Set Listing Style
2 Place Listing
3 Place Listing with Selection
4 Place Listing with String
5 Move Listing
6 Delete Listing
7 Get/Set Leadering Use
8 Get/Set Leadering Character

The Listing Element may be used to indicate hierarchical listings, bullet items, or simple table of contents. Listings are specialized paragraph elements within a Story.

Extract Operations

1 Get/Set Extract Style
2 Place Extract
3 Place Extract with Selection
4 Place Extract with String
5 Move Extract
6 Delete Extract

Extract Elements, sometimes also referred to as "pull quotes", are specialized text treatment within a Story that may contain additional adornments. Like a Special Paragraph or Listing Element, the Extract exists only within a BodyText Story.

Footnote Operations

1 Get/Set Footnote Style
2 Insert Footnote in Story
3 Delete Footnote
4 Get Footnote Story
5 Footnote Numbering (Page, Section, Document)
6 Get/Set Footnote Key Numbering Style (Arabic, Upper Case, Lower Case, Roman, Roman Lower, Asterisk)

The Footnote Element is attached to a marker references which is inserted at a given Story position. Footnote Styles are global to the Section.

Title and Headline Operations

1. Get/Set Style
2. Set Style for All in Section
3. Place Element
4. Move Element
5. Delete Element
6. Get Story Field(s)

Titles and Headline Elements are object treatment of textual document pieces which contain their own frames, adornments, and design intelligence. Titles and FixedHeads are attached to a Page element, while Heads and Subheads reside inside Stories and move with the text flow.

Graphic Operations

1. Get/Set Graphic Style
2. Set Style as Fixed/Floating
3. Place Graphic
4. Move Graphic
5. Delete Graphic
6. Get/Set Wrap-Around Image
7. Get/Set Image Wrap Offset
8. Adjust Graphic Element Frame
9. Flip Horizontally
10. Flip Vertically
11. Object-Link

The Graphic Element holds and image and provides Pages placement, style, and frame intelligence. Image wrap is available for Graphics that are fixed to the page, and do not fully impinge on flowable text areas. Frame adjustments are mapped to the closest designer allowable size. The PRO API may provide Object-Linking limited to graphic files. The fixed/floating option only applies to in-column Styles.

Caption Operations

1. Get/Set Caption Style
2. Attach Caption To Element (creation)
3. Delete Caption
4. Get Caption Story.

A Caption Element must be associated with another Element, and not all Elements will accept a Caption. Failure to attach on create indicates Element rejection.

Rule Operations

1. Get/Set Rule Style
2. Set Style Fixed/Floating
3. Place Rule
4. Move Rule
5. Delete Rule
6. Set Rule Orientation
7. Set Rule Span (In Column, 2-column, Full Width, Vertical)

The Rule Element is a graphic object used to partition space on a page on in a text flow. Rules that are In-Column may either be fixed or floating.

Border Operations

1. Get/Set Border Style
2. Attach Border
3. Remove Border
4. Get/Set Inset Amounts

A Page corresponds to the physical page entity. It is used as a canvas on which fixed elements are placed, managed, and effects the text flow of the document.

Page Operations also include modifications to Page Element attributes such as gutter, and side orientation.

Column Operations

1 Get/Set Column Style

Column operations are limited to selecting style choices. Control of magazine flow linking will not be supported in the first API release.

TypeFace Operations

1 Get/Set BodyText Font
2 Get/Set Headline Font
3 List Designer BodyText Fonts
4 List All Available BodyText Fonts
5 List Designer Headline Fonts
6 List All Available Headline Fonts

The Typeface Element manages the appearance of text font within a Section.

BodyText Operations

1 Get/Set BodyText Style
2 Get/Set Typesize
3 Get/Set Leading size
4 Get/Set BodyText Paragraph justification
5 Get/Set BodyText Hyphenation
6 Get/Set BodyText Paragraph Breaks and Minimum Lines

The BodyText Element manages the appearance of all standard text proportions and layout within a Section.

Running Element Operations

1 Get/Set Running Element Style
2 Get/Set Numbering style
3 Hide All/Show All Running Elements
4 Show Recto/Verso Elements
5 Get/Set Start Number
6 Get Story Fields

Running Elements are applied throughout a Section and share content changes. Recto/Verso operations apply only to Header and Footer.

Emphasis Operations

1 Apply Emphasis Style to Text Range
2 Apply/Remove Lining (underscore, strike through, double underscore, double strike through)
3 Apply/Remove Caps (All, Lower Case, Title/Proper, Small Caps)
4 Apply/Remove Color (Foreground/Background)
5 Apply Ink to Text to Text Range

Emphasis Elements are used to create a distinction between text or text-based elements (Listing, Extract, Special Paragraph).

The first release of the API supports application of Ink Styles to text ranges only.

13 Get/Set BodyText Font
14 Get/Set Headline Font
15 Get/Set Body Text Style
16 Get/Set BodyText Typesize
17 Get/Set BodyText Leading
18 Get/Set BodyText Justification
19 Get/Set BodyText Hyphenation
20 Get/Set BodyText Paragraph Breaks and Minimum Lines
21 Get/Set Header style
22 Get/Set Footer style
23 Get/Set Folio style
24 Get/Set Page style
25 Get/Set Column style
26 Get/Set any other default style choice

A Stylesheet is a collection of style choices occuring in a section or sections of a document. Stylesheet modifications effects the look and feel of a document within the constraints of a given set of design choices.

Story Operations

1 Select Range
2 Select All of Story
3 Delete Range
4 Insert String
5 Insert Special Character
6 Insert Symbol
7 Insert Extended Character
8 Insert Column Break
9 Insert Page Break
10 Insert Variable
11 Import into Story
12 Export Story to file
13 Apply Emphasis Style to Range
14 Apply/Remove Lining (underscore, strike through, double underscore, double strike through)
15 Apply/Remove Caps (All, Lower Case, Title/Proper, Small Caps)
16 Apply/Remove Color (Foreground/Background)
17 Toggle Bolding
18 Toggle Italic
19 Toggle Underline
20 Toggle Strikethrough
21 Toggle Superscript
22 Toggle Subscript
23 Find String
24 Replace String with new String
25 Insert Floating Element
26 Insert Floating Element with Selection
27 Insert Floating Element with String
28 Move Floating Element
29 Delete Element
30 Set/Get Story Name
31 Get Page for beginning/end of Range

A "Story" is Pages object-oriented treatment of Text which contains a rich set of text properties, expressions, and other Pages objects, which may in turn contain Stories. Story level operations apply to both the text and objects which may be manipulated inside Stories. The creation of Stories themselves is governed by other Pages objects.

Page Operations

1 Add Page (Magazine Flow only)
2 Delete Page (Magazine Flow only)
3 List Fixed Elements
4 Place Fixed Element
5 Move Fixed Element
6 Delete Fixed Element
7 Get/Set Page Style
8 Get/Set Page Sides (One, Two, Book-sided)
9 Get/Set Page Gutter Size

Appendix A, Operations

What follows is an outline form of the API functionality by grouping.

Session Operations

1. List available Design Models
2. List available StyleSets
3. List available Templates
4. Set default Design Model, Styleset or Template
5. Quit

Session level operations effect all actions between application start and termination, and essentially equate to user preferences. Choosing default Design Model, StyleSet or Template effects the creation of new documents only.

Document Operations

1. Create New Document
2. New Document from Template
3. Open Document
4. Save Document
5. Save Document As
6. Close Document
7. Use Design Model
8. Is Document "Dirty"

Document operations are levied against a document object that corresponds to a single Workspace Manager Object (e.g. traditional disk based files).

Pages does not intend to support remote file-system navigation at this time. This need will be better served by a solution available for general purposes.

Section Operations

1. Add Section
2. Move Section
3. Delete Section
4. Set/Get Section Name

A Section is a document division characterized by common stylistic applications. Section level operations are those which effect the organization, manipulation, and attributes of a document section. This includes stylistic elements that are always managed collectively within a given section.

Stylesheet Operations

1. Create Stylesheet
2. Delete Stylesheet
3. Copy Stylesheet
4. Set/Get Stylesheet Name
5. Set/Get Stylesheet Comment
6. List Available Stylesheets
7. Apply Stylesheet to Section
8. List Available Style Choices
9. Set/Get default Style Choice
10. Set Typeface Style
11. List Designer Suggested Fonts
12. List All Available Fonts

The project plan is to scale development in phases.

The first phase is internal only, and consists of a test server and client able to perform basic document load/save operations, text manipulation and attribution.

The second phase extends the server and shared PRO library to include floating and textual elements. The first draft documentation is prepared and early beta is possible. Pages filters may begin using PRO technology.

The third phase extends the API to include section manipulation and attribution (i.e. section creation and manipulation, stylesheet creation and manipulation, any remaining 'global' text operations such as header, footers, folios, column and page style). The loadable server is bundled with Pages by Pages and connected to appropriate methods. An updated server, client, and documentation will be available for beta.

The fourth phase extends the API to include page-based operations (fixed elements and page manipulation).

The fifth phase is to extend testing, review feedback, and make modifications as appropriate. Final documentation, the Pages server, PRO library, and client source examples are delivered.

8.0 Documentation

The target user of the API is an MCCA developer familiar with NEXTSTEP tools and appkit programming. Pages should be able to provide smooth integration to anyone familiar with modifying NEXTSTEP example source.

Pages will provide example code similar in format to the NeXT developer examples. It is anticipated that this code will function as the foundation for creating remote Pages by Pages clients. The source will include comments explaining how the APIs are used in conjunction with the Pages by Pages application.

There may be additional operations or services that would make the API more useable from the client perspective. These might include navigational functions, query functions, or document status.

5.0 Limitations

The Pages by Pages server will not provide basic file manipulations which can be performed in Unix. When the D.O. host runs a dissimilar OS, a file-manipulation PDO server may be an appropriate solution on behalf of the MCCA implementer.

Support for column linking, custom inks, and custom paragraphs is not planned for the first release of the API.

Session preferences will be managed separate from Pages preferences. This ensures that a remote connection does not reset user preferences.

Support for cut/copy/paste is not planned. The issue is two-fold, and stems from the global effect of the operation. First, cut or copy effects other applications on the same machine (login session), replacing pasteboard contents and interfering with user operations. Second, multiple Pages clients would contend for the shared global pasteboard, invalidating programmatic control. Still, implementation will be dictated by customer needs, and may include restrictions if made available. The API contains sufficient flexibility without cut/copy/paste for operations within Pages.

Nearly all processes between client and server will be synchronous.

6.0 Release and Compatibility Considerations

The APIs will be field upgradable by using NeXT bundle technology and Pages extension registration (shipped in release 1.0).

The APIs provide a layer of isolation between Pages by Pages internal operations and manipulations on the client-side virtual document. Pages by Pages product revisions will not adversely effect API clients.

The API release may not coincide with release of the Pages by Pages product.

7.0 Implementation and Delivery

The Pages Remote Objects and Virtual Document Kit consist of several components. The first is a executable library and set of headers that allow a client program to instantiate and manipulate document pieces. Overview and class documentation is provided with the PRO library. Commented code examples are provided with the library.

The second component is the Pages server itself, which runs as a dynamically loadable bundle in Pages by Pages.

Session operations effect operations between application start and termination, and equate to user preferences. The API will support setting default Design Model, Stylesheet, or Template choice for new document creation.

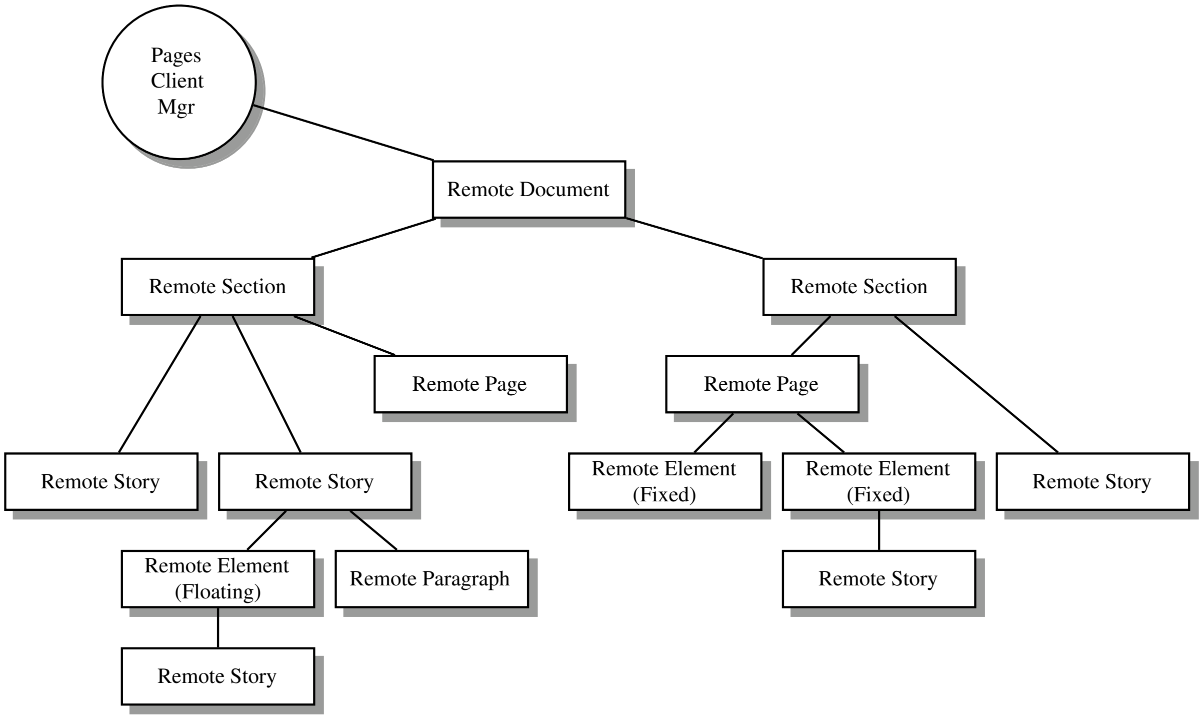
Document operations are those which effect the creation and existence of a top-level Pages object which corresponds to a single Workspace Manager object (i.e. Unix file or folder). This includes opening, creating, saving, or closing a Pages document; and changing or listing available Design Models.

A section is a portion of a Pages document that shares default stylistic choices. Section operations are those which effect the organization, manipulation, and attributes of a document section. Sections may be manipulated by creating, moving, or deleting. Functionality for changing section attributes include section name, stylesheet operations, and section-wide elements such as typeface, bodytext, header, footer, folio, page, and column styles. Stylesheets may be created, applied, copied, listed, annotated, or deleted. Stylesheets also allow default styles to be selected.

A story is the basic Pages text object, which supports embedded elements and a wide array of text treatments. The API includes the standard text manipulations such as string insertion, deletion, selection and query. It also supports insertion of column-breaks, paragraph-breaks, symbols, special characters, and variables. Story text may also be imported or exported to or from a file. Story text may be given special treatment by applying an emphasis style, or embellishments. The bold, italic, underline, strike-through, superscript and subscript embellishments have simplified UI. API exist for adding, moving, or deleting floating elements (Pages objects that live within the text stream). Query methods also exist to locate text within a story.Text may also be queried for position information

A page is the document output surface on which Pages renders information. Pages are usually created and deleted as necessary, but may also be added or deleted via the API. All fixed element operations such as add, move, or delete are specified in page-based coordinates.

Besides the selection of a given style, element operations apply to the nature of the specific element. Page API manipulate the gutter, whether a page is one-sided, two-sided, or book sided arrangement. Font operations include listing all or designer specified text or headline fonts. Body-text operations include changing type and/or leading size, justification, hyphenation, and paragraph breaks. Title, fixed head, head and subhead allow style choices to be applied to all like elements, or the selected element. Graphics may flipped horizontally or vertically, image-wrapped, or object-linked to a file. Folio, header and footer elements allow numbering style to be specified. Header and footer also allow variables to be inserted, and recto/verso placement. Border operations allow tailoring of border inset/outset amounts, and border removal. Rule element APIs allow change of orientation and whether rules are fixed or floating. Table operations include add/delete row, show/hide table title, label, or summary, cell justification and borders. Listing API allow tailoring of the leadering and leadering-character. Footnote API effect numbering type and sequence, and footnote placement.



Document Components in Pages Remote Objects

The components reflect a logical and familiar division of a document into smaller pieces, with the addition of object properties available on text itself.

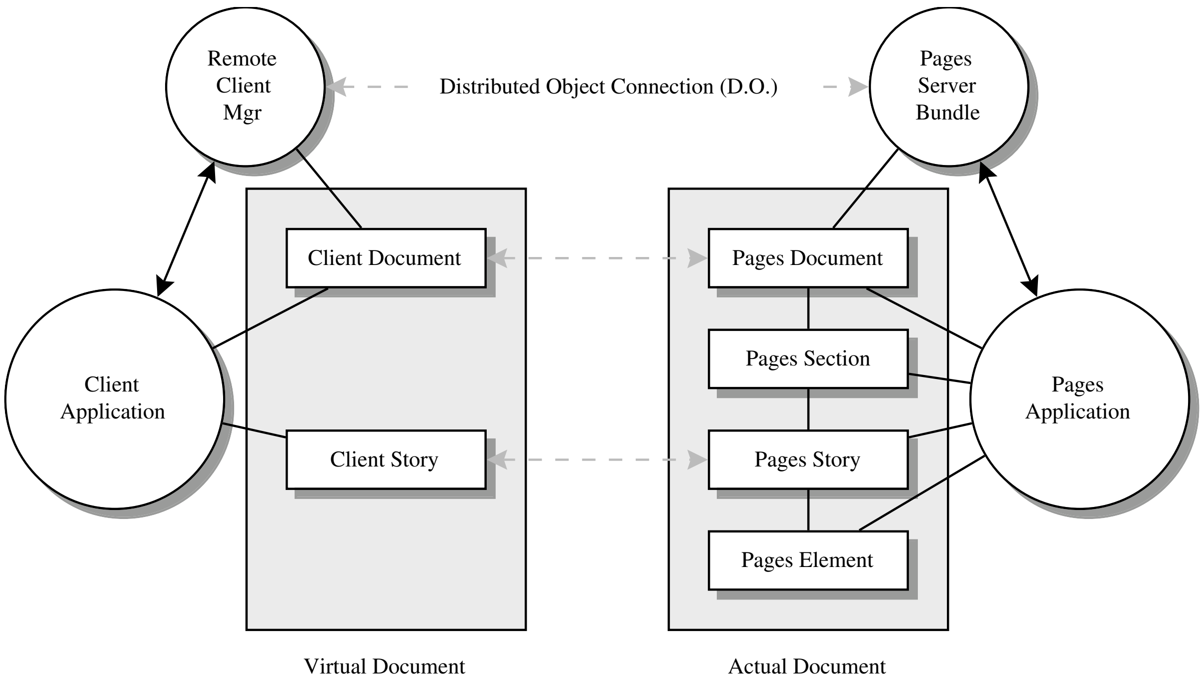
Objects on the client side are actual Object descendents, not Distributed Object proxies. The Distributed Object connection itself is maintained by a brokerage protocol in the Pages Remote Object superclass, and is entirely invisible to the implementor. This greatly simplifies client coding considerations, and allows customization of *any* PRO document component by the client via subclassing. For instance, the client may wish to create a subclasses of the Pages document which contains database information. The PRO technology also allows modular replacement of the distributed object conduit itself for future considerations.

Pages Remote Objects are precisely the same components used by the Pages extensible filter system for document mapping. The difference is that both the Pages Server and Pages Client exist in the same process space. This same technique may eventually support Pages plug-in extensions as well.

4.0 The API Set

A overview of the Pages Remote Object API functionality is presented by group. These groups correspond to the using the product from the user’s perspective.

3.0 Pages Remote Object Components



The API mechanism

The fundamental components of the Pages Remote Object API are the Pages server bundle, and the remote client object. The server bundle provides registry services and manages how remote objects are vended. The remote client object initiates the server/client connection, manages the session, and tracks vended pieces.

The virtual document itself is a collection of objects that represent components of a Pages document. Manipulating these components changes the structure or content of the document in the Pages server. The virtual document collection consists of a document object, section objects, story (text) objects, page objects, and Pages fixed and floating element objects. Operations that are specific to a class of element, such as graphic, are methods only found in that class. Operations that are general to all elements, such as moving a graphic or title, are grouped in a general element class.

A user API approach divides behavior into operations that make a selection, and operations that act upon that selection. For example, one action might select a range of text, while another copies that text to the pasteboard. Like the user interface for the program itself, there is always a single position, or focus, at a time which accepts input. The meaning of an operation may change depending on the location of the focus.

A user approach assumes there are actions to make a selection, then act upon that selection. The API is divided into messages that query, detect and choose a subject which becomes the focus for an orthogonal set of messages which modify the selection. Here we really mean "selection" in the broadest sense, where there is a positional location or "focus" from which we derive the target of a given action. Although this would appear a very profitable approach, we quickly encounter limitations by the absence of UI and user. The action of dropping a new Pages element is a good example: the mouse is tracked and a constrained placement rect is displayed in real-time until the user selects an appropriate drop point. The user has the advantage of visually inspecting both the placement rect and the relation to other existing elements on the page. While constrained placement information can be wrapped in the API, it would be difficult to easily correlate relations to existing elements. In fact, it would place a heavy burden on the client implementation for performing what would otherwise be a simple user operation with any degree of control and certainty. Furthermore, as the application’s user interface evolves, the API must change as well, even if the underlying architecture remains constant.

The functional API approach is characterized by the explicit presence of subject, verb, and target; for example, "add this Head to the given Story before the 4th paragraph". The set of allowed operations may be greater than that offered the user via the interface since we are not limited to a one-to-one mapping between UI and API. In fact, a functional API attempts to organize operations by intent. One benefit with the functional approach is that we can increase the specificity by adding arguments to the message. Of course, this can become cumbersome for involved operations, or even augment the complexity of simple operations. Another drawback is the the number of functions may be large to encompass the specificity required by each function.

With the advent of Distributed Objects (D.O.), new consideration can be given to both the API set itself, and how that API set may be used. A significant advantage is that calls no longer need to be invoked against a single target, as is necessary with all traditional APIs. Instead, calls are levied against appropriate objects such as documents, sections, or stories. This allows the API set to be segmented into smaller, more manageable groups which are more readily learned and applied. Secondly, it significantly enhances the ability to identify document pieces, since those pieces simply become client objects vs. function parameters. It also allows the API to be expanded without greatly impacting existing client code as new Pages classes become available.

Pages Remote Objects enhances D.O. by providing the ability for clients to subclass remote objects. Pages believes this better suites the need to customize objects in the MCCA environment. The remote connection itself is entirely invisible to the client implementation, simplifying useage. The result is a set of classes that may be subclassed and used expediently on demand, much like the appkit itself.

1.0 Overview

The *Pages Remote Objects* comprise the method of creating and orchestrating documents programmatically from an external process. In the most basic sense, this addresses the role of the traditional application programming interface (API). In an evolutionary sense, it provides the client with a *virtual document construction kit* filled with a rich set of intelligent elements. The client assembles pieces as per normal Objective-C programs, and the server automatically creates a Pages document with the identical structure.

2.0 Background

The traditional approaches to programmatic application control have involved macro or function calls against some form of port, or event control of the application's user interface.

The simplest form of GUI program control, event journalling and playback, has proven unsuitable for anything but simple automation, since positionally dependent environment changes effect the success of macro playback. It is also frequently the case that event-based control is disturbed by other user or system events that occur during playback, introducing synchronous playback issues. The greatest limitation for event-based, programmatic automation is the lack of substantial query ability. Therefor, event-based control is best suited for short, automated sequences that can be visually verified.

Intra and Inter-application macro languages have offered a richer set of control and verification mechanisms, because they may tie directly into an application. Many, like Visual Basic, are founded in well-known programming languages. Others are simplified or streamlined forms of algol-languages such as Pascal. Often, these macro languages present of limited set of data types in order to be accessible to a wide variety of users. This either inhibits, limits or complicates the ability of manipulating rich data inside these languages. Also, macros are typically interpreted to avoid requiring a separate development environment, making them somewhat slower executing than native code. Still, macros may be suitable to a wide variety of medium complexity automation and program control tasks.

The native language API functions (e.g. "C" calls) may provide a level of application integration that far exceeds the possibilities offered by macro languages. For one, the API may tie more closely to the underlying implementation, eliminating the translation layer required for the macro language. Second, a native language API can offer exchange of rich data such as structures, bitfields, memory and/or operating system data. The native language API may offer more complex program control mechanisms such as multiple threading or function callbacks. The native language API is suitable for high-volume, rapid executable systems often found in mission critical application development.

An API set may be implemented using various strategies. Perhaps the most straight-forward approach is to mimic the operations of the user interface. A second approach offers a set of "mathematical" functions, in that they operate on a given domain and range. They are distinguished by the implicit or explicit subject in any action (or verb).

Revision History

April 29th, 1994

ksb - Added Appendix with segregated lists of API functionality. Document published to Marketing and distributed.

May 20th, 1994

ksb - Appendix operations added and/or refined. Revision history and TOC added.

June 16th, 1994

ksb - EXPO edition. Appendix for example API use. Updated schedule outline, illustrations, limitations, implementation and delivery components.

Contents

Pages Remote Objects
(the Pages API)