From: Darryl Rubin
Sent: Thursday, October 22, 1998 5:46 PM
To: Scott Cottrille
Subject: RE: Annotations

The Hyperactive Desktop.doc

Personal Viewers Memo.doc

Each of these two docs has sections that discuss topics related to annotation, including ways of creating them (various kinds of note taking, esp. in the viewers memo), viewing and indexing them, etc. Unfortunately since the memos cover broader topics you'll have to hunt for the stuff that pertains specifically to annotation-related functions.

---Original Message---
From: Scott Cottrille
Sent: Thursday, October 22, 1998 5:17 PM
To: Darryl Rubin
Subject: RE: Annotations

Hi Darryl, we spoke with you a couple weeks ago about annotations. I'd like to read the memos Bill mentions below, where can I find them? Thanks,

-scott

---Original Message---
From: Yoram Yaacovi
Sent: Thursday, October 22, 1998 4:34 PM
To: Scott Cottrille; Marco DeMello; Jim Kott; John Scarrow
Cc: Bruce MacNaughton
Subject: FW: Annotations

FYI.

Yoram

---Original Message---
From: **Bill Gates**
Sent: Wednesday, October 21, 1998 7:44 PM
To: Darryl Rubin; Jon DeVaan
Cc: Bob Muglia (Exchange); John Ludwig; Yoram Yaacovi; Eric Rudder; Nathan Myhrvold
Subject: Annotations

I have always thought doing ANNOTATIONs right is a huge value.

I think there is some good work going on in the Dynamic Communities team. I would love to tout this as something big IF:

a) We knew how to scale it and get something out of it before others clone it
b) It related to our browser strategy and Office strategy.

I am concerned about our lack of a company view on this. Its an area where Office has to show leadership.

Office has 2 kinds of annotations already - annotations in the document and annotations on a web server. Having these be separate things is a great example of confusing complexity that people will never understand. However the web server stuff is not very flexible and no relation to the work going on in MSN.

There is also some work going on in Darryl's epad effort, and in research under Anoop Gupta and perhaps in the paperless office area. Someone smart needs to pull this together.

Office has got to pick someone to be forward looking in this area and help us decide how to relate their work to the MSN work and how to make it acceptable.

The best writing on annotations and its importance are the memos written by Darryl which there has been very little to no followup on.

Annotations are a good example of something where we want a base level of functionality in the browser but a much better level of capability to come in with Office. It gets easier when we have a strategic store that can help (storage+ perhaps starting with platinum?).

—Original Message—
From: Yoram Yaacovi
Sent: Wednesday, October 21, 1998 11:18 AM
To: Bill Gates
Subject: Innovations

Bill,

I am now on the ebook team under Dick Brass and Steve Stone, and I'm aware of the talks about showing off the RBG striping innovation at Comdex. It's really cool.

However, a little less than a year ago, when on the Dynamic Communities team, we created an innovation in a different space: the Web. The innovation allows people to post annotations to any web site, and let other people view these annotations. As far as I know, no one has done this yet, and besides being a new and cool user feature, it involves solving significant scalability issues (build a server infrastructure that will get hits for every Web page you browse to, if you have the annotations browser bar open). I know that you've seen demos of it.

There are some behavioral problems with annotations (people can abuse it), but they are solvable. And it's an innovation at the level of ICQ. It's working, and you can go to http://community to download the latest (M2) client. We only recently started the patent process, so this might be an issue.

I thought that this might be another option for us to demonstrate our innovation.

Yoram

Deeper Web/Shell/App Integration: The HyperActive Desktop

1 Introduction

In this memo I want to talk about extending the efforts we've made for unifying the shell and browser and suggest how we can take these ideas further; how we can:

- Further integrate apps and web browsing into the shell frame to achieve a navigation/viewing experience that is both richer and more seamless
- Better exploit push model to reduce information overload
- Simplify information access and filing by support of unified storage
- Use rich linking to make collaboration and sharing inherent and convenient features of the shell.

A couple trends motivate these ideas. One is that the PC is becoming a device more for communicating, collaborating on and consuming content than it is a device just for producing it. The other trend is that the web is becoming the primary metaphor for these consumption and collaboration activities.

The threat to Microsoft is that companies like Netscape and Lotus will be able to offer web-centric "desktops" that users will prefer living in and using as their launch points because they better support the user's web-centric metaphor.

This is not hard for our competitors to do, because the amount of local desktop functionality they need to subsume isn't that great (the Windows desktop isn't super rich, especially in the consumption/collaboration departments) and it's easy to design a set of HTML pages and controls that put a pretty nice face on the existing local desktop operations. This is, after all, partly what we are doing in Memphis.

The danger of competitive web-centric desktops is that they will be sold as platform-independent shells, shells that have an especial affinity for Java apps over native ones. This threatens the core of our platform and application strengths.

The challenge for us, then, is to enrich the desktop in web-oriented ways so that it is not so easy to replace, and so that running Windows-native apps under the Windows-native shell—especially Office—results in compelling advantages compared to Java apps on a platform-neutral shell. The web/shell integration

work we're doing in Memphis is a good first step, but there is much more we can do.

In this memo I'll describe how I think a set of next steps for evolving the shell UI for deeper web/shell/app integration and for richer exploitation of hyperlinking and unified storage. I call this next step, somewhat jokingly, the "hyperactive" desktop, for the way it exploits web integration and hyperlinking features. I don't suggest this to be our official marketing term for the new feature set! But for want of a better term I'll use it in this memo.

The hyperactive desktop builds on the ideas laid out at the start of this memo: richer shell/browser app integration, push-centric UI, unified storage, and rich linking. These ideas have been covered to varying degrees in several of my earlier memos, including "Storage Unification and Webification" and "Beyond Browsing" from last year, and "Unifying the User's Navigation/Viewing Experience" from 1995.

What I want to do here expand on this material, with focus on the UI and user experience as opposed to infrastructure. Where necessary I'll summarize key points from the earlier memos but otherwise will avoid duplicating material. I especially encourage you to read the Storage Unification and Beyond Browsing memos for much more detail and infrastructure discussion on those topics.

A couple other notes:

- While I cover a lot of material in this memo, my intent isn't to set forth everything I think we need to think through for a comprehensive "next UI". Particularly, there is already other UI thinking being done, such as Eric Michelman's work on web UI, which dovetails nicely with the ideas discussed here.
- The proposals in this memo are undoubtedly more than we'd do in a single release, and given the nature of UI design, people will find some ideas they dislike. I believe what I outline here is quite amenable to release subsetting and to alternative choices at a UI design level. My main goal is to express what concepts and features the UI should embody, and I'm less concerned about how closely the detailed realization matches what I say here.

2 *Capsule Intro to Unified Storage*

Some of the UI enhancements I'll discuss in this memo concern ways to exploit unified storage. For those who haven't read the storage unification memo, here's a summary of the storage model proposed there. The storage memo contains much more information on the storage features, api's, implementation, and how I envision unified storage being used.

Note that much of the UI discussion in this memo does not depend on unified storage; many of the features I propose are valid even in time frames where unified storage doesn't exist, although implementing them is made much easier when unified storage is available.

Unified storage enhances the file system to have these features:

- Three kinds of objects
  - Folders
  - Files
  - Links
- Properties
  - All file system objects have properties: folders, files, and links
  - OLE property model (multiple property sets with properties)
  - Network-wide property indexing and query
- Uniform container model
  - Folders can have storage streams in addition to children
  - One storage stream of a folder is its content: the HTML view of its contents
  - Files can have children (links, other files, and in the richest implementation, even folders)
- Rich linking
  - All forms of links, including HTML hyperlinks and OLE links, represented as explicit file system links
  - Links can be stored...
    - Internal to (children of) the file or folder containing them
    - External (stored separately), allowing links to be added to unwriteable objects
  - Separate source and destination anchor properties
  - Multiple link types (hyperlinks, annotations, responses, footnotes...)
- Advanced storage features
  - Read/unread tracking for files, folders, links, and annotations (the setting of read/unread is managed by the UI, not the store)
  - Single instancing for efficient message and app resources storage
  - Scripting of document and folder objects for customized storage behavior
  - Replication
  - Whole file caching (Coda)
  - Online backup/restore
- API's
  - Win32 file calls supported compatibly
  - Full set of COM api's based on OLE property and storage interfaces, with extensions

Unified storage creates opportunities to simplify and empower the UI. First, it lets objects of all kinds be treated the same way. Documents, web pages, mail,

newsgroup messages and appointments are all files. File folders, schedules, mail folders, newsgroups, and web sites are all folders. This means the same UI actions for creating, deleting, copying/moving, viewing, editing, and adding comments to documents will work for all these objects. It also means that the various kinds of objects can be stored together in the same folder, or returned as hits in the same search.

The UI need no longer have separate "clients" for documents, mail, schedule, web, and so on. The shell's folder "app" becomes a universal client.

The uniform container model also means that files and folders can be treated in much the same way. Both folders and files have content and children, and so both the content and children of either kind of container can be manipulated via common UI mechanisms. (The content of a folder is the authored view of its children, whereas the children of a file are its attachments and embeddings.) The uniform container model also makes it possible to have a completely seamless web view, where everything looks and acts like a (potentially editable) web page.

Having a uniform way to represent and store objects means that a common set of mechanisms can be used for security, replication, caching, backup/restore, querying, and so on. This simplifies not only the infrastructure but also the UI, because now a common set of UI mechanisms for these tasks will apply to many kinds of objects.

Finally, the rich linking features of unified storage make it possible to apply links in a very broad way. Links can be added into the objects they link or stored separately from either object; the latter case lets users link and annotate objects they can't write and also maintain personal webs of links and annotations that are not seen by others.

Links can also be used as a general mechanism for implementing annotations, responses, and discussions, meaning that these become shell operations that are common across all kinds of objects, including folders.

See the storage unification memo for a much more detailed discussion of the linking and other storage features.

3 *User Scenarios*

In this section I'll walk through a rather extended scenario of using the hyperactive desktop, interspersed with some design explanation as required. While I cover quite a bit, I don't intend to cover all the features of the hyperactive desk or to explain how they all work (I'll cover more of this in subsequent sections). The themes I intend to bring out are:

- A personal newsletter or a related metaphor based on the push model can become a primary tool for delivering and organizing a user's information. It attacks the information overload problem by giving the user a way to overview, orient, and focus.
- An enhanced, "frameless" shell frame can make navigation and viewing between and within documents more seamless, more visually attractive, and less cluttered. The shell experience becomes more document-centric, with less distinction between "shell" versus "app," "window" versus "in-place object", "web/internet" versus "desktop/local world".
- Rich search, viewing, and linking mechanisms make it possible to find, organize, assimilate, and share information more easily, and with less regard to type. Users can perform the same set of operations, the same way, on diverse kinds of information (documents, messages, appointments, folders, newsgroup threads, etc.) Most importantly, objects of diverse types can be collected together the same folder, meaning users are free to organize and view information based on topic instead of type.

As in the Memphis Active Desktop, the hyperactive desktop is just an HTML document.

So, the desktop is an authorable page that can contain arbitrary HTML elements, including embedded frames showing other HTML pages. It has task bars, menus, and toolbars that can be docked along screen edges or free-floated. And it has the desktop icons.

Compared to Memphis, there are a number of generalizations and enhancements in how all the various desktop elements are treated and how richly they can be authored. I'll cover these differences in detail later; the main point to make for now is that Memphis desktop's background and icon layers are combined into a single, fully authorable layer that can host both icons and embedded frames; the UI differences between background and foreground manipulation and behavior are eliminated.

Prominently featured on my desktop when I come in to work is a thing called my Personal Newsletter. This will typically be one of the windows in the desktop. It may well take up most of the space on my desktop, at least to begin with.

The newsletter is basically a personalized web document. The first page is like the front page of any newsletter—it has a short table of contents in one column and other columns for the highlights in major categories of interest: Hot Mail, Next Meetings, Top To Do's, Your Projects, Related Projects, Company and Industry News, Interesting Articles and Reports, and so on.

Because it pulls together so much information in an attractive, easy to skim and easy to drill-down form, the newsletter is probably the major center of my focus for much of the day. From it I can

- Scan the hot mail items and link to them or to my entire inbox
- See my next few meetings and key to-do items for today and link to my calendar
- See a summary of recently changed project documents and status information with change abstracts, and link to those documents of interest
- Review recent company and industry news capsules and research abstracts and link to the entire full text

The newsletter is also fun to read, because it is typographically nice and makes effective use of eye candy (graphic images and backgrounds). It may even contain a few pertinent cartoons or other diversionary material. Like any good publication, the front page provides an overview of the entire contents—including lead paragraphs from the key "stories"—with supplementary pages (linked from the front matter) giving more detail on each topic. This makes it easy to skim and prioritize my focus.

For example, a front-page "story" about a project document change would have the abstract of the change (taken from the document), plus a link to the changed document. (If a lead is too long to fit in a paragraph or two, it is spilled into a link page, or scroll bars are available.)

The newsletter is actually pretty smart about what information to show me. This is because it knows a lot about me and my company. It knows my position, the company org chart, what projects I work on or follow, what the project workflows are, and so on. It also knows the topic areas (keywords and concepts) I'm interested in, plus what topic areas each project is concerned about. It has learned some of my needs and interests from specific subscriptions I've made, but others it may have deduced through heuristic and feedback mechanisms I'll discuss later.

Whenever a file or database changes, my user profile tells the storage system enough to let it rank that change for its relevancy to me and update my newsletter if appropriate. (If the newsletter carries cartoons, it will even know my favorite cartoonists.)

Because it draws from a world of changing information on the corporate intranet and the internet, the newsletter is a dynamic document. To keep the changes orderly, it will update itself at regular periods (morning edition, noon edition, afternoon edition) with backlinks to previous editions. (Any given "edition" is, internally, just a list of information links, which the newsletter engine dynamically formats according to an authored template.) The edition backlinks could be shown on an "index of past issues" page of the newsletter.

So, I use the newsletter to orient myself on the day's work and to link me to the detailed information I need in the course of the day. Think of this as Outlook recast to be topic- rather than type-driven. Because the newsletter is both a push-

- Scan the hot mail items and link to them or to my entire inbox
- See my next few meetings and key to-do items for today and link to my calendar
- See a summary of recently changed project documents and status information with change abstracts, and link to those documents of interest
- Review recent company and industry news capsules and research abstracts and link to the entire full text

The newsletter is also fun to read, because it is typographically nice and makes effective use of eye candy (graphic images and backgrounds). It may even contain a few pertinent cartoons or other diversionary material. Like any good publication, the front page provides an overview of the entire contents—including lead paragraphs from the key "stories"—with supplementary pages (linked from the front matter) giving more detail on each topic. This makes it easy to skim and prioritize my focus.

For example, a front-page "story" about a project document change would have the abstract of the change (taken from the document), plus a link to the changed document. (If a lead is too long to fit in a paragraph or two, it is spilled into a link page, or scroll bars are available.)

The newsletter is actually pretty smart about what information to show me. This is because it knows a lot about me and my company. It knows my position, the company org chart, what projects I work on or follow, what the project workflows are, and so on. It also knows the topic areas (keywords and concepts) I'm interested in, plus what topic areas each project is concerned about. It has learned some of my needs and interests from specific subscriptions I've made, but others it may have deduced through heuristic and feedback mechanisms I'll discuss later.

Whenever a file or database changes, my user profile tells the storage system enough to let it rank that change for its relevancy to me and update my newsletter if appropriate. (If the newsletter carries cartoons, it will even know my favorite cartoonists.)

Because it draws from a world of changing information on the corporate intranet and the internet, the newsletter is a dynamic document. To keep the changes orderly, it will update itself at regular periods (morning edition, noon edition, afternoon edition) with backlinks to previous editions. (Any given "edition" is, internally, just a list of information links, which the newsletter engine dynamically formats according to an authored template.) The edition backlinks could be shown on an "index of past issues" page of the newsletter.

So, I use the newsletter to orient myself on the day's work and to link me to the detailed information I need in the course of the day. Think of this as Outlook recast to be topic- rather than type-driven. Because the newsletter is both a push-

or on the surrounding background of the desktop. Menus, toolbars, and window controls are auto-hidden; I access them by moving the mouse pointer close to the relevant window edge (if I want I can set all window frame elements to unhide together, or to stay locked on, or even to free float outside the window).

Using the spinner on the Intellimouse, I can view and scroll documents without ever needing to see the window frame at all.

This "frameless" desktop is more attractive and legible than a traditional windowed desktop because it is less cluttered. It also simplifies things by using the same control frame design for embedded objects as for windows.

With its seamless support for many information types, you might think I'm saying that the shell bundles most of Office. Not at all. What it mainly does is provide the homogenous framework into which Office functions comfortably slot. If you don't have Office, you can still navigate, view, and edit, but with a much poorer feature set (think Wordpad versus Word, for example). Again, the analogy to Wordmail is apt, but now, it is all of Office and not just Word that serves to augment the shell.

But, unlike today's Office, I no longer think of running apps or opening documents. I just link from one place to another.

Even the idea of File Open essentially goes away. File Open just links me to a navigation page with (authorable) links into the storage system. (Think the My Computer window, except prettified with HTML and with possibly some context dependent links into the user's storage.) I can link forward from the File Open page to the thing I ultimately want to open, or go "Back" if I change my mind. Being a case of navigation, it's modeless.

Some new features of the shell make this process of navigating around and going "back" much easier.

To start with, common operations like Search and File Open (by default) open a vertical split pane in which the search or file open contents appear. This pane, called a navigation pane, works much like the hierarchy pane in the file explorer and the search pane in Memphis. You can click on links in the navigation pane both to navigate through the search results and to open views of the selected item in the main display frame. This is also rather like the internet newsreader's vertically split view with threads on the left and message content on the right.

Navigation panes are good not only for search and file open, but also for document and workbook outlines, site maps, and any other kind of navigational schematics, from storage hierarchy views to graphical topic maps and VR landscapes. The TOC column of the Personal Newsletter is (by default) in a

navigation pane. All of these are things that have links you want to keep handy for switching among a set of related pages or documents.

You can also manually split the main display pane, which opens a navigation pane on the content under view so that you can use any of its internal links for navigation. For example, opening a navigation split on "Fred's Favorite Links Page" would let me keep Fred's links in view in the left pane so I could quickly explore them without always having to go "back back back" to Fred's page.

Multiple levels of navigation split could be made to work, but I won't address that here.

Anyway, I'm delving through my morning newsletter, now in the Your Projects section and checking out a document that has come back to me from review. I have the newsletter TOC in the left split pane and the Your Projects page in the right. High up in the latter's Recent Changes column is an unread item, "My Modest Proposal" with the notation, "Has received comments". I click on the document link and the document opens.

I want to peruse the review feedback, so I choose the Comments (i.e., annotations) view from the view menu. This opens a view similar to Word's Comments view, in that it has a contents pane and a comments pane, but there is also a third pane: a navigation pane showing the document outline, and having comment nodes displayed as leaves. This makes it easy to navigate the document and to check out where the comments are.

The lower split pane containing the comments is called a tracking pane, because it tracks what's shown in the content pane. Tracking panes can show a variety of things, including comments, footnotes, link previews (lookaheads), and graphical topic maps.

I'll admit that with three panes up, the document window may look a little cluttered to some people. There are a few things I could do if this bothered me. I could of course close either the outline or the comments pane. (Remember, hovering the pointer on a comment will still display it.) Alternatively, I could set one or both of these panes to auto-hide. Or I could detach them to free float.

For these various options, think "Memphis taskbar", which has the same kinds of options, all available via drag/drop of the frame itself. In the hyperactive desktop, we generalize so that any navigation or tracking pane can be made to hide, auto-hide, free float, or dock along any window edge. In fact, taskbars then become an instance of these generalized panes; the taskbars are accessory panes of the main desktop frame.

The three-pane view is nice because it makes it easy to navigate through a document's comments sequentially or in any order in relation to the document

outline. Better still, I can change view settings on the outline pane to filter on various response criteria. Each comment node is an object with properties, so I can apply all the normal viewing operations here. For example, comments include properties for the responder's name and the type of comment (question, correction, suggested change, issue, etc.). So, I can filter for just remarks for a given person, or just questions or issues; or I can categorize the comments so that the outline groups all questions together, all issues together, and so on.

One other point about comments: unified storage lets us implement comments via links, meaning that they needn't be contained inside the document itself. People can comment on documents without having write access to them.

Everything I've said about comments applies to the other kinds of items you can view in a tracking pane—footnotes, hyperlinks, etc. The same kind of three-pane view will show any one or a combination of those objects.

Well, I've read my newsletter and my first major task for today is to do some research on biometrics. I'll need to pull together some information on this topic, boil it down some, and run it by some coworkers for discussion.

I use the shell's search button to open a pane containing a search form.

The search form gives me ways to conduct searches spanning a variety of scopes: within the current content frame (document or view) only, or within other physical scopes like local machine only, intranet only, nationwide, or worldwide. For my search on biometrics I'll do a worldwide search.

I want to include in my search literally anything I can get my hands on that relates to biometrics: documents on the subject, email messages, newsgroup threads, web pages. The search pane probably has a way to specify the types of information to cover, but let's assume that in this case the default is to cover all types.

Since I'm searching a scope outside of the previous document, the search results come back both as the current document in the content pane, as well as a hit list in the navigation pane. This is quite useful, because anything in the content pane is a document that can be edited, annotated, printed, saved, and so on.

What I get back from the search is a results document—actually a folder of search hits (links), but because folders are HTML documents, it can look nice; for example, it can be a tabular display providing information (properties) about the hits, including possibly a thumbnail. Suppose I get back lots of hits; I'll now want to do some fancy viewing to cull through and narrow down what to look at next. For this, the shell gives me all Notes-like features for sorting, filtering, categorizing, and aggregating on properties in a list, as well as for calling up predefined, named views that are supplied by the container (in this case, the search folder).

These viewing features will work for any list, table, or hierarchy, in any pane. They're broadly useful, since much information is presented this way, whether it be lists or hierarchies of documents, document sections, spreadsheet ranges, mail and discussion messages, links, comments, footnotes, appointments, and so on. The review document example above showed where it would be useful to use viewing features in a navigation pane (the document outline with comment nodes).

In my biometrics research I'm specifically interested in eye tracking, so I'll filter my search results for those keywords, sort by relevancy rank, and subsort by date. Next I'll categorize the view on Subject/Title to see if any patterns emerge there; then, say, on item type, so that things like email messages, documents, and web pages get grouped with their own kind—makes it easier to scan what the mail and documents are about. Then I'll put the view into thread view to crawl through some of the mail and newsgroup threads.

Again I'm navigating between different kinds of information within the single shell frame. There's no need for secondary windows to open unless that's what I choose. Throughout this process, the use of a navigation pane to hold the search results makes it much easier to explore and refine the hit list and check out the hits.

OK, so I've found a selection of items of different types I want to keep, I'll copy those to a new folder, "Biometrics Research" (or simply delete out of the search results folder the items I don't want).

As I review some of the found documents, I annotate them and make referential cross links for later reference. A single drag/drop mechanism lets me make links between documents, or between anything, including mail messages, newsgroup items, appointments, and web pages. The shell lets me treat all kinds of information uniformly. Also, it doesn't matter if I'm making links in documents I have no write access to; the shell is creating external links for me in those cases.

I find a few things I want to respond to, and not just some of the email and newsgroup messages. For those cases it's obvious: I select "Reply" on the context menu and get a properly addressed send note. But I also want to respond to one of the documents I read. "Reply" in this case brings up a send note addressed by default to the document's author, and with a link to the document automatically inserted.

Alternatively I could add comments to the document as I read it, and the author would be notified of the comments via their personal newsletter. The author could then comment on my comments, with my newsletter notifying me about those, and we'd end up conducting threaded discussions within the body of the

document itself, via chains of comments. (Section 6 discusses this collaboration scenario in more detail).

Next, I want someone else to review the entirety of this research. I drop the whole research folder into a mail message. Before sending, there are some comments I want to make on this collection of material, which I do by annotating the folder the same as I would a document. This is possible because folders, too, are HTML documents, and so support comments, footnotes, etc. (Think of it this way: the web view is now editable.)

On the other end, the recipient can open the folder from within the email message, or drag it elsewhere in their storage hierarchy. The annotations I made will show up in views on the folder.

Perhaps some of the links I mailed to my colleague are to documents in my own storage tree. This is OK; assuming this is a Coda world, all my files exist on servers for which share points are automatically established and user-level permissions are in force. When I mailed the links folder, the mail system checked the recipient's permissions to the documents referenced and where necessary (should be rare) asked me if it's OK to grant the needed permissions. I never need to do anything a priori to share stuff; mailing a link is all it takes.

The mail also checks whether the recipient lacks connectivity to my storage tree, if necessary attaching the referenced documents to the mail message.

Well, this has been a long scenario. Time for lunch?

To recap, the key features illustrated in this section were:

- Use of a dynamic, push-model "newsletter" that overviews the day's work and acts as a convenient point from which to drill down.
- Seamless navigation and viewing through information of all types within a single shell/browser window.
- "Frameless" windowing for less UI clutter and more uniform treatment of linked and embedded objects.
- Ability to organize and store information without regard to type (a folder is a folder is a folder)
- Availability of rich Notes-like viewing features for any collection.
- Shell/browser enhancements via navigation and tracking panes to make it easier to navigate and view information, and to unify some currently disparate navigation/viewing features.
- Use of rich linking, threaded views, and intelligent "reply" features to make sharing and collaboration easier.

The rest of this memo will go through these features and others in more detail.

4 The Desktop

In this section I mainly want to cover how the hyperactive desktop differs from what Memphis has done. This is not meant to be a complete discussion of the desktop features and what enhancements we'll want to include.

Memphis provides the first step toward a web-centric desktop by introducing an HTML background that can contain frames showing other HTML pages. The background is basically a window that has been made full screen and frameless, and which can contain embedded windows.

Several of these backgrounds can be open at once, each on its own nontransparent Z-plane (thus, only one is visible at a time). One of these is the default desktop background, and the others are called "channels". App windows that the user opens exist on their own Z-planes, so they can be displayed on top of a background (desktop or channel) plane.

The rough spots in Memphis are that frames on an HTML background are treated differently from normal windows. Nontraditional handles are used for moving and sizing them, and they cannot be minimized. If you click on a link in a background frame, it opens a new window that is not part of the background (you cannot navigate within the same window frame). You switch among the channels using a special channel bar.

There are also a lot of desktop elements that cannot be authored via HTML, including the icons and task bar. These are each implemented as separate non-authorable frames: a transparent overlay for the icons, and a dockable, auto-hideable frame for the task bar.

The hyperactive desktop makes all these elements HTML-based and eliminates the special treatment of the background planes and their embedded frames.

As in Memphis, the screen will support multiple background Z-planes, plus a plane per floating window. Unlike in Memphis, the background planes aren't special in any way in terms of what you can display there. You can expand any window into a background plane, meaning that it takes on a frameless, fullscreen appearance.

The main thing you want in a background plane is the default desktop. This is implemented as a folder, much as it is in Memphis. The children of this folder are the various objects that are "on" the desktop. In addition, the folder has a stream of HTML content that defines the authored appearance (the web view) of the desktop. You can of course display other folders or other documents as background planes too. These would be the equivalent of Memphis channels.

So far this probably sounds similar to Memphis. What's different is that channels are no longer anything special, they are just normal folders or documents that have been expanded to the fullscreen, frameless display mode. Also, the desktop icons (the children of the desktop) now have a fully authorable appearance. Rather than sit on a separate plane, the icons are actually part of the desktop background, and the author and decide any placement and visualization that's desired.

Designing the desktop folder's appearance—or the appearance of any folder for that matter—works this way: you open the folder's web view and use normal HTML authoring tools to design its content. You then drag objects into the view, which adds them to the folder. By default, the dropped items will visualize as normal icons at the drop coordinates, but you can now change this—editing the icon image and properties to change its appearance and behavior. (These edits actually change properties on the child object.)

Why is this useful? Because you can line up the icon with other background elements. You can make its style consistent with the rest of the background. You can make it a button or other control rather than an icon.

Another thing you can do is make the child visualize in-place as an open frame. How? Again, just by editing the properties of the object, which include a set of visualization properties (see Section 5.2.4). This creates the equivalent of a Memphis background frame. However, you can now do this for any kind of child object; and do it just by diddling properties rather than using a special authoring mechanism. You can turn an open frame back into an icon the same way.

And, the user can do this at runtime, expanding icons into frames and vice versa; it's not just an author-time thing.

Moreover, because there's nothing magic about a folder being in the background—it's just a view where you make a window fullscreen and turn off the frame—everything I've said about icon authoring and open frames applies to the normal, in-window view of a folder too. That is, any folder can have a fully authored view, with authored icons and open subframes.

Think of this the full realization of the "folder as document" model, where like any other document, a folder has an authorable content that can include in place objects. The open frames are those objects. One implication of this is that we can use the same move/resize UI for both the in-place object and the desktop frame cases; we no longer need a special one for the latter.

Not all folders will be manually authored of course. The web view for other folders will come from a default HTML template based on, say, the current Memphis web view of a folder. Another case that needs to be handled is that of a user dropping an object on a folder icon. Here, a child is being added but not

explicitly placed in the web view image. The software will need to compute a reasonable placement in this case. The folder's web view could have a region designated by the author for "unauthored" icons to be auto-placed, or the software could just look for any free background space starting in the upper left quadrant.

One problem with rendering embedded frames in normal folder views is that it can slow down operations like traversing the folder hierarchy. But this is no different from what happens when you're navigating a set of web pages, some of which have large embedded images. IE already handles this case—the solution is to render them asynchronously and allow further user input in the meantime. Folders, and documents in general, can work the same way.

This folder-as-document model answers a couple questions you may be wondering: First, do embedded frames show up in the task bar? No, just as frames embedded in documents don't. This behavior is also consistent with what Memphis decided. However, the document itself—including folders or documents expanded into background Z-planes—*will* show up in the task bar. A channel bar is no longer needed (though we may choose to keep it for other reasons).

The second question is, can you minimize an embedded frame? No, just as you cannot minimize frames embedded in documents. What you *can* do is change the frame properties to make it display in place in a different way, such as an icon. The notion of maximizing and minimizing only applies to things that exist as top-level frames (normal windows); i.e., to things that can appear in the task bar.

In Memphis, if you touch a link that is in an embedded frame, you get a new, floating window. In the hyperactive desktop, this doesn't have to happen; the link can navigate in place, and by default, does. If you'd rather get a new window you perform a UI modifier when you click the link (see Section 5.2.4). A given frame can also have properties set that change the default linking action to "new window".

As for icons, touching one of them will launch a new window, though again, this is a matter of properties and available UI modifiers that control that behavior (with the alternative being to navigate the frame that contains the icon to the new place—not a commonly useful option).

One problem with allowing arbitrary folders and documents into the desktop background is that some may be too big for the screen. That is, you may need to scroll the background. The best way to handle this is via auto-hidden scrolling controls. These could be hosted on the task bar and/or on bars of their own. Another option would be to turn on the background's frame, in which case the full complement of window controls, menus, and toolbars becomes available.

I said at the start of this section that other desktop elements like the task bar and channel bar are HTML-authorable elements. In fact, they are folders whose children are the objects they contain, and whose appearance is governed by the folder's HTML stream. (A nice byproduct of this design is that control panes become things whose contents you can view using normal folder views, including the ability to search them.)

With this kind of approach, the only thing that makes the channel and task bars special is that they run in special frames that are always accessible, no matter what's on the display. That is, these frames have the Memphis functionality that lets them dock along any screen edge, optionally auto-hide, or free float.

But even these features needn't be special-purpose. The shell can have another folder, the "screen" folder, in which screen level frames like these are hosted (as links or children). Properties on each such item would indicate its settings regarding docking/hiding/floating. It them becomes possible to associate arbitrary frames—for example, content windows and folders—with the screen and let them dock, auto-hide, or float.

To recap, the hyperactive desktop changes the desktop model from Memphis in these ways:

- The desktop as well as the task bar, channel bar, and channels are all folders.
  - Their appearance can be HTML-authored
  - Their child objects can be manipulated via normal shell UI.
  - They can be viewed with normal folder views and even searched
- Any folder or document can be set to display as a background plane; that is, as fullscreen and frameless. A new UI control will be needed for this ("super maximize").
- The appearance of a folder's children is fully authorable. They can be set to display as icons, buttons, or other UI elements. They can also be set to display as embedded (in place) frames.
- Embedded frames work like in place frames in a compound document. In fact, these are exactly the same thing, and use the same UI for move, resize, properties, etc.
- Every normal window, including those maximized as background planes, appear in the task bar. (Embedded frames don't appear in the task bar.)
- All frames can be set so that link navigation occurs in-frame or opens a new (floating) window.
- Nonembedded frames support properties that control their display behavior
  - Whether the frame docks or free floats
  - If docked, whether the frame auto-hides
  - If floating, whether the frame is always on top

5 *The Shell Frame*

The hyperactive desktop extends what Memphis has done for web/shell integration, both by integrating still more web functionality into the shell and also by extending this integration to applications as well. The gives the user a more seamless experience in navigating, viewing, and manipulating all kinds of resources: documents, web pages, mail, discussion groups, appointments, and so on.

It means that navigating from one kind of information to another, such as by clicking a link, never requires that new windows be spawned; as in a web browser, the linked-to information just appears in the current window (unless the user specifically wishes a separate window for it). The user can now focus exclusively on where they want to go and what they want to look at or search for, and needn't be concerned with differing tools or windows needed to accomplish those tasks.

Further, the deeper web/shell/app integration means that a fairly rich set of operations can be made available for all kinds of information; for example, a set of broadly applicable navigation and viewing options, as well as editing, printing, and collaboration functions.

You can think of the model for achieving this as being Wordmail++. That is, the shell will offer a frame with a bunch of standardized UI, and with shell/app frame negotiation to allow apps to fill the client area of the frame and augment the frame UI as needed. What is different from today's app/shell negotiation model is just that the standardized frame UI is richer, as is the set of things that can be negotiated.

That being said, it's also a goal of the new shell frame to avoid a cluttered UI, let alone a UI where a big percentage of the frame UI changes every time you click a link. The new frame therefore has features to address these problems.

Considering how much richer the shell frame becomes, you may wonder if I'm saying that we end up bundling much of Office functionality in the shell. No. What the shell frame does is provide the set of "compartments" into which the UI components of Office and other apps can slot. Windows itself would ship as it does today with a set of applets, like the standard browser, simple mail and text editor components, and so on. Office would replace or augment these as appropriate.

I'll now look at the features of the shell frame in more detail.

5.1 Shell Frame Elements

The shell frame accommodates all the elements you traditionally think of for an app. It hosts menus, toolbars, scroll bars, a status bar, and client (content) areas.

All these things are instances of one basic thing: a pane. All panes share these features:

- The have HTML-defined content
- They can be hidden or revealed
- They can be docked along any window edge, free-floated, or dropped into a menu, toolbar, or other control pane
- They can be set to auto-hide, with a variety of settings controlling the hide/unhide behavior (see Section 5.1.3)
- When floating, they can be set to be "always on top"
- Where applicable, they can be moved and resized

5.1.1 Control Panes

UI elements like menus, toolbars, and the status bar are called control panes. There can be any number of these, but of course the shell will define a standard set of them that forms the framework of the UI (e.g., a menu bar, toolbars for navigation, filing, and formatting, etc.)

Via the app/frame negotiation protocol, applications can dynamically create, inspect, and edit control pane contents and properties as needed to customize the frame UI. Typically, apps will just insert items into the existing, well-known control panes and add their own additional panes. This is basically like today's frame negotiation, but with more of the frame elements subject to negotiation.

Where applicable, users can customize control panes by direct manipulation of their contents; for example, by dragging around menu items and controls. Memphis has already begun to permit some direct manipulation of the Start menu, and Office has done some of this for toolbars, so all I'm saying is we combine these capabilities and make them general for all control panes.

5.1.2 Client Panes

Panes that show the content of documents and views are called client panes. There can be any number of these, but the typical maximum will be three. I won't discuss cases with more than this.

There are three kinds of client panes: content, navigation, and tracking panes.

Content panes are where views and document contents are displayed. Typically there is only one content pane, or else two if the user has split the pane.

Navigation panes are where things like search hit lists, document outlines, site maps, filing hierarchies, and other navigational aids are displayed. They are a generalization of IE4's search pane. Navigation panes are a kind of content pane that has the following special linking behavior. Clicking on a link in a navigation pane causes the shell frame to navigate to the linked content. This causes all panes *except* the navigation pane to update according to the content that was linked to.

There is one exception to this behavior: if you click a link that navigates within the scope of the navigation pane information (e.g., "Next 10 hits"), then the navigation pane updates but the other frames do not.

Together, these behaviors of links in navigation panes make them useful as tools for exploring material based on a set of search hits or on an outline or map of the material itself. Navigation panes are created either inherently as part of choosing a named view (see Section 5.3) or by using a splitter control. Their default presentation is as a vertical pane docked on the left edge of the shell frame (left split pane).

Tracking panes are a generalization of Word footnote and comment (i.e., annotation) panes. They are where things like footnotes, comments, link previews (look ahead), and context maps are displayed. A tracking pane behaves like a content pane that is sync'd to another pane; as the other pane is navigated or scrolled, the tracking pane will update to reflect the range of material currently shown in the pane it slaves from. The default presentation for tracking panes is as a horizontal pane docked on the bottom edge of the shell frame (horizontal split pane).

Tracking panes can display information in either of two ways.

- Range of items. The tracking view is a list of the tracked items that are visible in the tracked pane. An example is the Word footnotes and comments view, which displays multiple items at a time. This mode of display is, obviously, most appropriate where the items being tracked have modest size, and where the tracked pane doesn't have too many in view at once.
- Single item. The tracking view shows only one item at a time—the currently highlighted item in the tracked pane. This mode of display is most appropriate for hierarchy views, where the tracked pane shows a list or hierarchy, like the contents of a folder or discussion group. The tracking pane shows the content of the currently highlighted item.

A tracking pane can in principle track any other pane, even another tracking pane. Consider the example of a footnotes pane, where there are comments on the footnotes, and so you have a separate comments pane to show those.

While the infrastructure can easily support multiple panes tracking other panes in any arrangement, at UI level we need to simplify the capability to keep it comprehensible. There are several options from very restrictive to less so:

- Allow only a single tracking pane that can only track a content pane, and can only be shown if there is one content pane (i.e., content pane isn't split). This is the Word97 model.
- As for the previous case, but allow tracking when there are split content panes. The tracking pane will track whichever content pane has (or last had) the focus.
- As for the previous case, but allow multiple tracking panes up to some limit. (e.g., two).

I prefer the third option because I think there are cases where it's very useful to be able to track two kinds of entities at a time, e.g., comments as well as footnotes, or link lookaheads as well as a context map.

5.1.3 Pane Options and the Frameless UI

A goal of the shell frame is to provide a visually simpler and less cluttered UI for the user, even though it is trying at the same time to host fairly rich app UI controls and advanced features like navigation and tracking panes.

This is done by supporting options for how and where panes display—the docking/floating/hiding options listed at the start of Section 5.1. Basically, this does for panes in general what Memphis has done for the task bar in particular, and what Office has done to slightly lesser extent with its toolbars: implement direct manipulations to let the panes be dragged along any boundary, where they'll dock, or else outside the frame altogether, where they'll free-float; plus implement properties to control hide/show/auto-hide.

Obviously, by hiding nonessential control panes and auto-hiding others, you can get rid of a lot of UI clutter while still making the UI accessible when needed.

The ultimate UI simplification occurs by hiding all but the most essential control panes and making the rest auto-hide along the frame edge where they're docked. When this is done, you end up with essentially a "frameless" content pane. Content frames then look more like pieces of paper on the desktop than like windows, an effect which can be heightened by the right graphics design along the frame edges (i.e., minimal border and a very subtle, diffuse shadow).

Use of sophisticated visuals can augment this frameless effect even when certain control panes need to be revealed. The best example is scroll bars. We can make these semi-transparent, so that they don't completely obscure the area they cover. Floating control panes may also benefit from this treatment. Transparent versus opaque visualization could be controlled by a pane property.

The frameless presentation is ideally suited for the typical browsing scenario where all navigation happens either from clicking on content links and/or via Intellimouse operations. Who needs menus or even scroll bars in this case? It even makes sense for many "light editing" scenarios like creating and sending email, where everything can be done by the mouse and context menus.

With well-designed context menus, it should be rare to need menus and toolbars except for heavy duty editing. The few commands that a user may like to have handy could easily be put (even by the user) into a single small floating or auto-hidden toolbar.

I think we should push on the frameless UI concept very aggressively. It should ultimately be the default setting for all frames, and/or there should be a single UI command (context menu item or frame property choice) for putting frames in this state. Of course, the default frame layout that a given user prefers should be taken from the user profile, so that users who want menus and/or toolbars by default can have it that way.

Now, it's clear how to do a frameless UI when the frame is full screen, such as a Memphis channel, but what about when the frame isn't full screen? Some cases are:

- Mouse pointer rests on the frame edge or moves very near it on the inside of the frame. The scroll bar appears.
- Mouse pointer crosses the frame edge to the outside and then rests very close to it. The auto-hidden pane on that edge rolls out.
- As above, but there are multiple auto-hidden frames on this edge; e.g., menu and toolbars on the top edge (this case also applies to screen-level hidden frames). Possibilities are:
  - Unhide them all
  - Unhide only one of them (e.g., the "outermost" one, or the most recently used one). For any other(s), draw thin "flaps" at the window edge that the user can rest or click on to unhide the chosen pane instead.
  - Unhide only one of them as above, but if the mouse doesn't soon move into the unhidden pane, sequence through the others until the user moves the mouse pointer.

To be really useful, auto-hiding needs to be more than a simple binary setting for each pane that says auto-hide versus not. Some cases are:

- When one pane unhides, other related panes should too. For example, menus and toolbars together; horizontal and vertical scroll bars together; or even the entire UI together. A property that lets a pane slave its auto-hide behavior to another (including bilaterally) would solve this.
- When multiple unrelated panes are hidden along the same border, the order in which they present themselves should be controllable. For example, you may want to impose an explicit unhide order, or specify that uhiding should happen in a most recently used order. A property that specifies a relative unhide precedence would solve this.
- Once a pane or set of related panes unhides, it should be possible to make it stay visible for a specified hysteresis period. For example, once menus/toolbars are revealed, you're likely to hit them more than once, so it may be desirable for them to stay visible for some period, regardless of where the mouse pointer is. They'd disappear automatically if they get no further interaction after the hysteresis period expires. Each interaction would reset the hysteresis timer.
- You may want the unhiding of panes to be triggered on some other event than that of crossing a window or screen edge. For example, perhaps the editing UI for an app should unhide when the user clicks in the client area of the pane. This idea combines well with the hysteresis parameter. The window starts out frameless for browsing (with auto-hidden scroll bars), but clicking inside the window unhides the menu/toolbar UI for editing, and this stuff hangs around until you've stopped editing. A property letting you specify the triggering event for hide/unhide behavior would solve this. App code needs to be able to hook into this logic so that it can raise its own hide/unhide events for the shell to respond to. For example, when a user clicks in a drawing, a toolbar specific to drawing may unhide.

We normally think of auto-hide being meaningful only for docked panes, but that's not so. Consider the example I just gave of unhiding a drawing toolbar when you click in a drawing. It's just as valid for this to be a floating toolbar as it is to be a docked one.

The properties described in the preceding list clearly offer a lot of control over how panes handle auto-hiding, in that you can set up the UI to respond in many different ways. I don't suggest that we expose such low-level properties to users, however. For the most part users should just deal with auto-hide decisions at a whole-frame level, where the choices are more along the lines of

- Auto-hide the whole frame UI versus don't
- When unhiding on an edge crossing, unhide the whole UI versus only along that edge
- Use "sticky" unhiding (hysteresis) versus unsticky

There's one other thing you can do with panes that I haven't covered above: you can drop then in control panes. For example, if you have a navigation pane

displaying a search form, you could fill in the form with some settings and then drop the pane on a toolbar. Pressing the resulting button would reopen the pane with the settings intact for you to complete that search request. Or, dropping a navigation pane set up to show a document outline would result in a button that would open an outline pane on the current document. You could even, say, drop a menu or toolbar into a toolbar; pressing the button would recall that menu or toolbar.

A note on direct manipulation: as we enrich the UI with more kinds of direct manipulation, discoverability and predictability becomes a problem. The user will wonder, what happens if I drop here? To solve this, I suggest that we implement balloon tips that will appear when the cursor rests over a drop point for some period.

5.2 Navigation and Search

In today's UI, the shell and apps implement somewhat overlapping navigation functions, sometimes with conflicting conventions. Each world also has features that would be useful in the other.

For example, Word97 provides a "Select Browse Object" control on the scroll bar which affects the behavior of the paging controls; it lets you choose whether the scroll buttons will navigate by page, heading, footnote, comment, and so on. This is much more flexible but also somewhat overlapping with IE's Prev/Next buttons.

The shell and apps support two other kinds of navigation: via hierarchy panes as in Outlook and the Windows explorer, and by clicking on links. Here, again, support isn't uniform across our apps or the shell (although it's been getting progressively better).

Finally, we have different support for search in different places. The biggest discrepancy is that our apps focus mostly on search features for searching within the currently viewed document, where as the shell (browser) focuses entirely on searching between documents.

The navigation and search features of our apps and the browser need to be integrated and made more flexible. The new shell frame attacks this problem by providing a standard set of navigational and search tools that application components can augment.

5.2.1 Previous/Next, Favorites, and History

The previous/next buttons are enhanced. They support the Word97 notion of "select browse object." This could be done via a button the way Word does, or

via a drop-down list in which the browse objects are named. Either way, the user should be able to see the current setting in or below the previous/next buttons.

The shell will come with a set of standard browse object choices, and apps will be able to augment this list via app/frame negotiation. The choices will include

- Word97 browse objects (or a subset, which Word augments). E.g., section, page, comment. Useful for navigating within a document.
- Chronological history. This is IE4's previous/next behavior, but with persistent history, so that you can go "back" to stuff visited in prior logon sessions. The history list records all visits, including web and local and intranet documents and folders.
- Topic. For content that contains topic links (see Section 5.5.2), this will navigate along the topic chain (i.e., documents that are the Top of a topic).
- Favorites. Navigates along the Favorites list.
- Search list. Navigates along the search hits from the last search (usually still displayed in the search pane).
- Navigation list. Navigates along the list of links in the navigation pane. Examples would include a filing hierarchy, document outline, mail folder, newsgroup, topic list, search list, or other list of links. The links are traversed in the order they visually appear in the pane, starting from the currently highlighted entry.

As you probably realize, most forms of previous/next navigation can actually be done by opening an appropriate list in the navigation pane and using the last of the above previous/next settings. For example, you could navigate by history or favorites just by opening those lists in a navigation pane and either touching links directly or setting previous/next to follow the navigation list. The advantage of using a navigation pane explicitly is that you can use the filter/sort/categorize options on the navigation view to control the order of previous/next traversal. Using the built-in favorites and history browse object choices is, of course, much easier.

One more point on previous/next: the design I'm proposing puts previous/next within a document on the same continuum as previous/next between documents. Partly I do this because the operations are fundamentally the same, and partly because is a web world the question of "within document" versus "between documents" is a lot grayer (in a set of related web pages, is a web page a document or a part of a document?).

Nevertheless, for some of the operations like previous/next page and previous/next in history, it may make sense to provide dedicated buttons in addition to the general ones that can browse by any selected list. Because you can direct manipulate control panes as discussed in Section 5.1.3, you can easily customize the UI to put the various navigation buttons where you want them, such

as in tool bars, scroll bars, and menus. You can have them show up in more than one place, too.

5.2.2 Navigation Panes

The use of navigation panes is a significant feature the shell frame provides for enriching navigation. This is because navigation panes provide an overview of the navigation context you are traversing, and because you can control the order of traversal by applying viewing operations to the pane, like sorting, filtering, and categorizing (see next section for details).

For example, you could open a navigation view on a newsgroup, categorize on thread, and traverse by message within thread. You could open Favorites, filter it on the keyword "news", and traverse just your favorite news. You could open History, categorize by site, and traverse by history within selected sites.

Another use of navigation panes is to display the logical topic hierarchy of a collection of material. The shell can draw such a hierarchy based on topic links if the material includes them (Section 5.5.2).

With full sorting, filtering, and categorizing available, you can make much more effective use of big lists like Favorites and History without needing to manage the physical organization of the lists.

Navigation panes are also important as viewers for search lists. They make it easy to view a search list, refine it, and visually organize it (filter, categorize...) in the manner most suited to exploring the search results. Because the navigation pane stays in view as you explore the hits, you can browse through the hits much more quickly.

The UI for presenting hierarchies in a navigation pane will of course need a full complement of outlining controls, such as for expanding and collapsing the hierarchy. Balloon text for the hierarchy items should also be supported (a la Word's outline pane in the Online view).

5.2.3 Searching

In our apps and shell today, we make a distinction between searching within a document and between documents. But it doesn't need to be this way. It's important to make search more uniform, because as I pointed out above, the web world blurs the distinction between "within" and "between" documents.

The hyperactive desk solves this by defining a common search form that applications can augment via frame negotiation. The key to this is a form field for controlling the scope of the search, which can be

- Current frame (i.e., document or view) only
- Desktop (desktop contents and currently open frames)
- Local machine
- Intranet
- Internet

The model for searching is basically the web model. That is, searches are done by going to a search page where you fill out the search criteria. But here, we take advantage of navigation panes to avoid needing to visually leave the context we're starting the search from.

So, a typical search in a Word document is a frame only search, whereas a typical IE search is an internet search. The unified search form makes the full complement of search options available across the range of scopes (of course some options could be grayed out depending on the scope selected, or the type of object currently being viewed).

Note that the ability to do in frame searches applies to all views, including things like traditional folder views. Anything you can view in a shell frame is content you can search within. This makes it much easier to navigate within large collections, like document and mail folders. You no longer have to conduct these searches visually.

Now, most intraframe searches are simple string matches and for this it would be pretty heavyweight to open a navigation pane with a big search form. Adding a "quick search" item to a menu or toolbar easily solves this. Quick search would open a small floating navigation pane with a subset of the search form (e.g., just the match string field, and all other search choices defaulted). So, the quick search could look basically like Word's currently find dialog.

Note that this quick search is really nothing more than a normal search with a canned set of form and pane settings. If we didn't supply it, the user could as well do it themselves by opening a search, diddling the form and pane options to create the quick search pane, and then dropping the pane into a toolbar (see Section 5.1.3). Indeed, a user can package up any number of canned searches this way for later use.

Note that today's notion of File Open goes away and is replaced by the search model. File Open—if we even provide a command of that name—becomes just a subcase of search; it is a search form tailored to finding files, and as such, also includes links for file navigation.

Because it is just a search case, the File Open form itself can be linked from the general search form, hosted on a menu or toolbar, and customized by applications as well as the user. Also keep in mind that the "result" of File Open is just to produce a list of links that you pick from; for example, a list of search hits, or the contents of a folder you navigated to. There is no notion in the UI that you are doing anything except navigating when you finally pick from that list.

5.2.4 Link Following

In a web world, clicking on links to go places is obviously a frequent operation. It's therefore something to focus UI work on. We have a few rough spots today.

First, we don't treat everything as links that we should. *Any* kind of reference should look and act as a link, including section, page, figure and other references. Word97 has made a start here.

Or consider mail. The names appearing in a mail header should all be links to the users' address book entry. The list of attachments should be links. Anywhere you see a file or a folder reference (whether textual or iconic) it should act as a link. When I say act as a link, I mean it is visualized as a link and the mouse cursor responds in the link-like way (hand pointer).

Second, for such an important feature, we don't offer much richness. We don't offer options on how to visualize links, or whether they should do in-frame versus new-frame navigation. We also don't do anything to help users know where links lead to; they have to just try them.

These things are all easy to fix by defining a set of properties that can be set on links and on frames. They include:

- Link action. Determines the action to take when the link is clicked. One of:
  - Do nothing
  - Navigate in frame
  - Navigate in new frame
  - Edit in place (i.e., OLE in place editing in nested frame; only applicable if link is set for in place visualization)
- Link visualization. Offers several options on how the link should display in context. One of:
  - Textual link. The property provides the text string and optional display style (underlined colored text, highlighted text, etc).
  - Icon. The property provides the icon ID.
  - Button. The property provides the button parameters and label.
  - Bitmap. The property provides the bitmap.
  - Anchor. When displaying as an anchor, the link provides no display content of its own, and instead attaches to content in the host document.

The anchor property provides the range information for the anchor. The visualization setting property provides the style for visualizing the anchor (highlight, enframe, color reverse, etc.)
  - In place content. The property provides the cached presentation, object frame settings, and activation information.
- Balloon tip. Provides optional balloon content to display when the mouse pointer hovers over the link. The shell can add information to the balloon to give the user a clue about how costly traversing the link will be (e.g., provide textual or visual feedback if it's a slow-link case, or a big fetch case).

All these properties can be set on a per link basis (usually by the author). Properties can also be set at a frame level to provide a default behavior for link action and visualization. For example, the system default is for in frame navigation and visualization as blue underlined text; however in a Word frame the default visualization for comment links may be set to be yellow highlighted text, and the frame may be set for new frame navigation.

While the link- and frame-level properties for new frame behavior will usually result in "the right thing" happening, there are times when the user wants the other behavior. To let the user know which way a given link will navigate, the hand cursor will display as just a hand for the in frame case, and as a hand with a drop shadow for the new frame case (or, as a hand surrounded by a little frame). A modifier like shift-click will let the user choose the alternative behavior for the link; thus, shift-click opens a new window if the cursor is a hand, or navigates in frame if the cursor is a shadowed hand.

5.2.5 Tracking Panes

Tracking panes also help users with navigation, because one thing they can track is link previews.

A link preview is a lookahead of the information being linked to by a link in the content pane. For links that have balloon content, this content is displayed in the tracking pane. For links that don't, part of the linked content is prefetched and displayed here.

Thus, by tracking link previews, users can get an overview of where the currently visible links lead.

Link previews are useful for more than just the web-like hypertext case. They're also useful for previewing any kind of hierarchy. This falls out of the way that unified storage treats hierarchy as a case of linking (see Sections 2 and 5.5). However, because hierarchies typically have lots of entries, each of which has sizeable content, the tracking pane for a hierarchy should default to a single-item preview (see Section 5.1.2).

Scenarios involving hierarchies include previewing the children of folders, news, and discussion groups, as well as address book listings. (Address books can be implemented as folders of entries in unified storage.)

Finally, note that tracking views can be graphical, not just textual. For example, the shell or an app could supply a graphical context view that displays a graph of topic nodes to portray the logical context surrounding the material in the content pane. The topic nodes could be based on topic links defined in the material (see Section 5.5.2), or any other internal knowledge of the material's structure.

5.3 Viewing

On a web-centric desktop, viewing is certainly one of the most important operations. The viewing features must be exceptionally rich, applicable across many kinds of information, customizable, and extensible.

The key features are:

- Full Notes-like viewing features in all list-oriented content
  - Sorting (multiple levels)
  - Filtering (simple and with boolean operators)
  - Categorization (multiple levels)
  - Aggregation (a categorization where an expression over the category contents is evaluated and displayed on the category heading)
  - Threading (a categorization based on a thread ID property that identifies the root object of the thread; the root can be any kind of object, not just a message)
  - Named, user defined views (combinations of above options with settings)
  - Named, container-defined views (backed by arbitrary app code)
- Viewing features are applicable to any pane
  - Any list-like information in a view can be sorted, filtered, etc.
  - However tracking panes are mainly filter- and thread-view only because the list order is driven by the pane they slave from
- A standard set of generally useful views; for example:
  - Normal
  - Page Layout (how view will print)
  - Outline (content pane categorized on level property)
  - Thread (content pane categorized on thread ID property)
  - Online (navigation pane with outline for navigation)
  - Comments (an outline pane and a comments pane; see Section 3)
- All views are searchable (see Section 5.2.3)
- All views are editable (subject to content-specific restrictions). This will be explained in the next section.

Except for external comments and links, I don't think any of the above features are particularly new. Notes, obviously, has had most of them for a long time; Exchange, Outlook, and Office have some of them too.

What's new here is that I'm saying we collect a rich set of viewing features we've implemented in a scattershot way across a number of products, rationalize them, and host them as basic features of the shell. Individual apps can then augment them as necessary, for example, by adding to the list of named views under the View menu, and by adding options to the view settings form and viewing toolbar.

Having a full complement of viewing features makes other features of the shell much more useful. For example, you can now filter, sort, and categorize things like the history and favorites lists. If these lists get big you no longer need to worry about organizing them into a hierarchy to make them manageable; with viewing, you can impose any dynamic organization you want to help you focus on what you need.

The ability to apply viewing options to any pane raises the complication that viewing controls like the view menu need a way to know which pane to operate on. The most obvious way to handle this is operate on the pane that has the focus. Alternatively, the view settings forms could have radio buttons indicating which panes to apply the settings to. It might also be handy to put an optional viewing widget on the pane frame that the user can touch to open the view setting controls. View settings could also be access from the pane's context menu.

One point worth highlighting is that with these viewing features, the shell is now providing a set of views that used to be the specific domain of the apps; for example the online and comments views. This is because the ability to navigate by a content outline and to see the comments on a piece of material are fundamental and applicable to all kinds of material; these shouldn't be app-specific features.

Also note that unified storage makes building some of this support fairly straightforward. For example, unified storage provides a standardized api for the shell to enumerate both the children of any container, which also include its internal comments and links, as well as comments and links that have been associated with it externally. Thus, the same hierarchy and table viewers that the shell uses for folders will work for the other kinds of content you want to put in navigation and tracking panes. Any content that can't be handled in this standard way can, of course, be supported by app-supplied code for that kind of content.

What set of standard views to define is an important question. In the bullets above I've suggested a few views based on those in Office, but that's just a starting point. I think we'll want to think through some additional views based on collaboration scenarios.

5.4 Editing

Our current UI practice is to create a rather large distinction between views that can be edited and those that cannot. This comes from the very different mindset between "shell" and "app". Traditionally we think of the shell as a viewer only, with all manipulations done via commands and dialogs. Win95 has made a tiny step away from this with directly editable file names.

Applications on the other hand have lived at the opposite extreme, where the assumption is that most views are *all about* editing. This is fine, except it throws a lot of UI in the face of people who only want to browse. I've already explained how we can use a "frameless" UI to solve this problem.

It's worth noting that even apps aren't totally consistent about editability. They have many accessory views that are mostly about showing and not editing.

A goal of the hyperactive desktop is to eliminate the distinction between things that can be edited and those that cannot (while keeping the world uncluttered for the people who are mainly viewing oriented.) In principle, anything you can look at is something you may at least want to be able to annotate and possibly excerpt and send to someone. You may well want to make other changes into the material itself, if you wish to use it in something else you're putting together (and assuming the material isn't copyrighted). There are many cases where this isn't possible today, or at least not easy.

The view editing features the new shell frame supports are:

- In list-like views, modifiable properties can be edited in place
- Columns can be rearranged and resized by direct manipulation
- New columns can be dropped on a view from a column well
- Dragging rows between categories in a categorized list will implicitly change the value of the property the view is categorized on
- A user can add comments and links to any material—even if the content is read only—without modifying the viewed document (external comments and links—see Section 5.5.4)
- If a user tries to edit the content of material that he has no permission to write, then depending on the situation, either a warning explains that saving will only be possible to a new location, or else the editing is denied (see below).
- A change in the frame visuals indicates when the contents of a frame has been edited by the user (the user will also know this when they try to close the frame or navigate away from its contents; see below).

The generalized editability of views has a number of implications. First, it says that we will be putting forth a standardized set of table editing conventions and

offering a standard table viewer. The table viewer would talk to the underlying data via a standardized interface like OLE/DB.

I suggest that we factor the table viewer into two levels of functionality, one that ships with the shell, and another that ships with Office. The basic table viewer would have all the key Notes viewing features, direct manipulation of columns, basic outlining/hierarchy features, and editing of field contents at a Wordpad level. Advanced table viewing, editing, outlining, and formatting would be supplied by Office and by other ISV application components.

For editing in table views to be meaningful, the underlying data source must cooperate, for example, by providing realtime field validation. While this isn't strictly required—the data source can always refuse the edit at field update time—I nevertheless expect that most sources will want to supply custom validaters. Fields in a table should be implemented as controls to make this possible (with the shell supplying a default control for this purpose). Thus, ISV's can override the standard implementation at both a field level, or at the whole table (view) level.

A second implication of view editability is that we need to come up with a save model with works in a world where navigating among documents happens in a single frame. Today we have at least two save models. In the shell we allow in place editing of file names, and for this we implement a persistent, no confirmation model. That means that all edits are implicitly committed as they occur, and without the user confirming. Office, by contrast, implements a nonpersistent, positive confirmation model, meaning edits are not persistent and must be confirmed to be saved. The question is, in a world where folders and documents are treated the same, what's the right save model?

There are several possibilities, but I suggest we consider a persistent, positive confirm model. Here is what this means. You're viewing content—it doesn't matter what kind—and you make some changes. These changes are persisted into your cache as you make them, either instantly or using an appropriate auto-save interval; but they aren't committed to the underlying data source until you explicitly save. You can do this yourself, or else the system will ask you at an appropriate time by bringing up a save dialog.

When should the system do this? It depends on what you were editing. The decision will be up to the data source via the frame negotiation protocol, and possibly under the influence of preference settings. Here are the likely defaults:

- For certain tabular content, such as a folder properties display, as soon as you complete editing on a field or row.
- For documents structured as a single container, as soon as you navigate away from the document.
- For documents structured as a set of linked containers (e.g., set of linked web pages), as soon as you navigate outside of the logical set.

- On any of several triggering events:
  - Immediately upon revisiting something you've made uncommitted changes to
  - When the uncommitted changes have been in the cache more than some time
  - When the cache needs room and wants to flush the changed content
  - You close the frame you were editing in
  - You logoff

Automatically popping a save form up can be rather annoying, especially if it's done at the wrong time, like too early. "Too early" is when you were navigating away from the changed document to consult something else, and you knew you'd be going back to continue editing.

A way to solve annoying save popups is to let the system agent handle the job. I say this because the agent is modeless; it can offer discreet suggestions without disturbing the user's flow, and get out of the way if it's clear the user has no interest.

So, say you navigate out of a document you just changed. The agent appears to the side of the frame and asks "Save the changes you made to Foo document?" Here, the name Foo is a link to the uncommitted document and the choices are Save, Save As, Discard Changes, Decide Later.

If you choose not to respond to the agent right now, fine, it will disappear after a few seconds, or as soon as it notices you doing much input (presumably in this case you know exactly where you want to focus right now). Picking Decide Later is the explicit way to make the agent go away.

Now, let's assume you've let a bunch of uncommitted images accumulate because you ignore the agent a lot. All's not lost, because uncommitted images are still in your cache, and you'll get more reminders at the other times mentioned under "Triggering events" in the list above. Another, subtler reminder is that the frame's border visuals will have a distinct emphasis any time you're in a frame with uncommitted changes.

Now for a question: Where do you end up when you navigate back into something that has uncommitted changes in the cache? For example, you've edited but not saved Foo and some time later follow a link to Foo. Do you see the original Foo or your edited one? This is a hard question and you can easily argue it either way. One thing is clear, though: if you imagine some program running that asks for Foo, it should get the original image, not the changed one, because the changes haven't been committed yet.

This suggests that a similar behavior is appropriate at the UI level, at least in some cases. But not in all cases: think of linking out of your changed document and

then Back'ing to it. One step away and back. You expect to see the changed image, not the original.

I think the answer is that we should do one thing or the other based on heuristic knowledge of what the user is up to. For example, are the changes super recent, is the user retracing a short navigation path, is this happening in the frame where the edits occurred? If yes on most or all counts, it's good bet you want to see the changed image. If no, probably better to show the original. But in the latter case, what if you really did want the changed version? I suggest that whenever the system presents an original document where an uncommitted version exists in the cache, the system agent can pop up and say so, offering to replace the view with the other image.

As you can see, the save model question is fairly complex and will require some further thought and experimentation.

A third implication of view editatibility concerns what it means to save in multipane views. The problem is that some multipane views involve content that isn't part of a single document, such as when you have a list of search hits in a navigation pane and a document in the content pane, and maybe even some link lookaheads in a tracking pane. You may well want to edit and save the search hits, or perhaps make edits in the tracking view, but now it's rather confusing about just what the current "document" is, and therefore, what File Save is going to save.

One possibility is to use the same approach as is used for viewing options on panes. That is, make save operations apply to the pane that has the focus. Let me make clear what this *doesn't* say. If you have, for example, panes showing a document's contents, its outline, and its footnotes, then saving when the outline pane has the focus doesn't do some weird thing of saving just the outline someplace; the document is saved because that is what the outline pane is a view on. On the other hand, if you are saving when a search pane has the focus, then it is the underlying search folder that is saved.

To make this approach at all workable, the save form would need to make it very clear exactly what you are saving.

An alternative way to deal with this problem is to restrict what panes can be edited, so that a navigation or tracking pane can only be edited if it is providing a view on the same container as the content pane. If it is showing something else, then the pane cannot be edited, and so the saving issues don't arise.

5.5 Linking

If the hyperactive desktop has a fundamental operation, it is linking.

As I've already discussed, linking is the way that essentially all navigation happens, whether touching an icon, browsing a web of hyperlinks, or navigating a hierarchy. Anywhere that a container reference occurs in the UI, in any kind of view, it should render and behave as a link. And I use the term container in its broadest sense, meaning any folder, document, message, newsgroup, address book, and so on.

Even any appearance of a user reference in the UI should be treated as a link to the user info relevant in that context (i.e., it should be a link to the user address book entry or a specific part of it, like the email address).

Links are also the way that most or all UI is invoked, especially if we follow through on the web UI model being proposed by Eric Michelman.

Indeed, the UI controls in menus and toolbars are links. Menus and toolbars are nothing more than containers that render their contained links in a particular way and provide menu-like and toolbar-like interaction behavior. You could as well drag one of these elements to the desktop (or into any folder), in which case it becomes a regular looking link that happens to invoke a UI command.

I've already said quite a bit about navigating using links, so in this section what I'll focus on is the process of creating links and what features are available for that.

Note that everything I say about links applies to annotations as well. Annotations, such as Word comments and footnotes, are just a macro operation for creating a new object and then linking it to another.

The basic features of links are that

- They can be created by direct manipulation as well as by menu operations
- They have properties, including a type, source/dest anchors, and others
- They are stored as full-fledged objects in the storage system, meaning they can be
  - Queried, filtered, sorted, categorized, and aggregated on their properties
  - Read/unread tracked (meaning link followed or not)
  - Protected via access permissions
- They can be stored external to the objects they link
- They can link anything to anything (i.e., anything for which you can make a URL)
- They can visualize in a number of ways, including as various forms of hypertext object and as in place frames (see Section 5.2.4).
- They can update and/or cache themselves along a spectrum of hot to cold, as determined by their update properties

5.5.1 Creating Links

Links can be created in the ways you'd expect: via direct manipulation and context menu actions, as well as via drop-down menu and toolbar commands. (Remember, any command can be put in any or all of these places.)

Because links can now have properties and source/destination anchor ranges, some refinement of current UI is needed. For example, you need to be able to highlight both the source and destination targets when establishing a link. The source end is easy: if the user initiates a drag operation from inside a highlighted selection, use that as the link anchor. If the drag is not inside a selection, establish an insert point at the drag origin and make that the anchor.

A similar approach can be used for the destination anchor. If the drop point is inside an active selection, make that the anchor. Otherwise establish an insert point at the drop coordinates and make that the anchor.

Menu and toolbar actions to make links would work in a two-part way. The user would establish a selection or insert point and call up a Link or Insert Link verb. The cursor would change to graphic indicating that a link is in progress. The user would navigate to the target selection or insert point and use a menu or toolbar item Complete Link to finish the job. (I prefer this to saying that click implicitly completes the link, since the user may need to navigate to get to the target, and that means some clicking.)

It may be desirable to let the user fill in link properties as part of the link creation operation, versus first creating a link and then manually opening the properties page. This could be done by automatically calling up the properties page as a floating pane when the source anchor is established. The link properties page would stay in view throughout the link creation process, so that the user can review and set the properties at any time. There are two good reasons for doing it this way:

1. To let the user manually fill in the destination anchor if, for example, the destination is something they can't directly navigate to.
2. To let the user specify whether the link should be stored with the source or destination container, or external to either (see Section 5.5.4).

Once a link has been created, its properties can be called up and changed at any time by right clicking on either end of the link.

5.5.2 Link Types

Link type information includes two orthogonal kinds of type information:

- Containment type. Conveys physical hierarchy or logical hierarchy with deep copy semantics.
  - Parent
  - Child
  - Hyperlink (AKA shortcut)
- Topic type. Conveys logical (topic organization) hierarchy.
  - Generic
  - Next (in topic)
  - Previous (in topic)
  - Next Topic
  - Previous Topic
  - Topic top (head of a topic)
  - Footnote
  - Cross reference
    (Document, section, page, paragraph, line, table, figure, chart)
  - Comment

The containment and topic type of a link are usually set by the shell or the active app based on the UI operation that was invoked. For example, Insert File creates a child link with a generic topic type, whereas Insert Footnote creates a child with a footnote topic type. Footnotes, cross references, comments, and other items would have choices on the Insert menu to make creating those kinds of objects easy.

The containment type of a link is used by the shell mainly in synthesizing hierarchy views. For example, a node that has child links can be expanded to show the children, whereas a node which is a cross link would act as a leaf in a hierarchy view. And, when displaying a container in the content pane, a link to its parent will be included and rendered as a parent link.

The topic type of a link is used by the shell mainly for navigation and viewing operations. For example, by filtering a tracking view on the topic type, you can choose to see only comments or only footnotes, or both. Or, when viewing a container full of links, you can filter so that only the top (head) of each topic in the collection is shown.

As for navigation, by setting the previous/next buttons to navigate by topic (next/previous topic links), you can move between the root pages of a set of topics, skipping the detailed pages about each topic. Conversely, selecting to navigate by previous/next in-topic links will move you sequentially through the material in each topic.

The inclusion of things like explicit previous/next topic and in-topic links is up to the content author. When not present, the shell uses a default ordering where possible (or else disables that navigation option). For example, in absence of explicit in-topic links, the shell will navigate between peer items in the same container. In absence of topic links, it will navigate through history along document/web page boundaries.

5.5.3 Link Properties

Links have other properties besides type. The other standard link properties include:

- Basic file system properties (timestamps, permissions, GUID, etc.)
- Source and destination anchors
  - Identity of object anchored to (e.g., pathname or GUID)
  - Range in object of the anchor
- Link action, visualization, and balloon tip (see Section 5.2.4)
- Link update behavior
  - Prefetch/caching settings
  - Hot/warm/cold update settings

The combination of these properties allows for a very rich set of linking scenarios, including the full gamut of traditional hypertext linking, in place editing, and hot linking scenarios. All these cases can be turned into any of the others just by adjusting the link properties.

Moreover, linking can be as fine grained as desired, because the separate source and destination anchor properties allow for differentiating what specific part of the source is linked to what part of the destination. This lets the shell visualize the source and destination sides of a link appropriately, as well as navigate to the right destination spot when a link is followed.

5.5.4 External Links

The hyperactive desktop supports the notion of external links. What makes this possible is that links are represented as file system objects and contain separate source and destination anchor specifications. Together, these things mean that links have a representation that is independent of the items they link.

There are actually three places where a link can be stored relative to the items that it links:

- As a child of the source object. This is the traditional web link case.
- As a child of the destination object. A good example is responses in discussion folders (newsgroups), where a given response links back to the thing it responds to.
- As a child of some other container. This is the case of a true external link, such as a link made between documents on a CDROM. The link itself is stored in a folder stored someplace on a hard disk separate from the CDROM.

Remember that when I say "link", I also include all forms of annotation. Thus, annotations can be external as well.

The shell and apps determine where to store a link based mainly on what operation the user is performing. For example, when responding to messages or other documents in a folder, the default is to add the response as another document in the same folder, with the new document containing the link to the item it's responding to. When adding comments inside a document, the default is to add the new comments as children of the document if the user has write access. When making links between things the user has no write access to, the default is to store an external link in a default per-user folder allocated for this purpose. The user can change or override these decisions via a link's properties page (Section 5.5.1) and via global preference settings in the shell.

The question of where to store external links bears more discussion. While in principle they can be stored anywhere in the storage hierarchy, I think that for starters we should adopt a simple convention: If a link cannot be stored in the source or target document, then an external link is created and stored in a particular folder: a per-user, "private web" folder.

What this folder represents is all the user's personal links—links created as part of the user's own view of the world, and not seen or shared by other people.

By default, the web will render in all views the user's private web links as well as the links that are contained in the material in question; but the user should have the option of turning the private web on view and off. This could be one of the view settings, along with those that control the other viewing options I've already discussed.

The enabled/disabled setting of the user's private web could also guide the shell in deciding whether to store newly created links as private or public in cases where that's ambiguous (such as when the user does have write access to the document).

Note that because a private web is just a folder, users can if they wish open the web folder itself to view and manage it. For example, they could categorize on document name to find all links associated with a given document and then delete

those or else click on them (or open a tracking preview pane) to inspect what they link.

5.6 Other Functions and Frame Seamlessness

From what I've presented so far I hope you've gotten the sense that the standard shell frame presents a structure that can support a rich variety of navigation, viewing, editing, and linking functions. By modeling the UI organization for all this stuff after an evolution of Office, we can make it very natural for Office to slot in and augment the basic shell features.

Please keep in mind that what the shell supplies as standard features may only be a subset of what I've talked about so far, the rest being left for Office and other apps to slot in.

I'd like to talk about a few more categories of function where the shell can provide richer functionality than today while also offering a better framework for seamlessly hosting app (especially Office) extensions.

First, the shell should have an Insert menu, providing a standard way for inserting objects into any context. Apps would extend this menu to include their specific types.

So, to create something new, you would use either a template (perhaps sitting on the desktop) or File New to create a blank document or folder, and then use Insert to insert specific kinds of objects.

Unlike today's world where you need to choose the kind of container you're creating at File New time, we could possibly simplify the list to a few generic types; for example, document, folder, link. This is possible in a timeframe when all our documents are HTML native, meaning they share a common representation. So, you just create a document and start typing.

If there still need to be major file structure differences required by different apps at the outermost level of the document file, then the app itself could reformat the document the first time an Insert of that app's type is done. That is, the first Insert would dynamically establish the document's outermost format. Any future inserts would result in embedded objects in the OLE sense—the outer document would not reformat.

Undo at app and shell levels should also be integrated and be made persistent. For example, suppose I make some changes to a document, then link to another document and make changes there. I should be able to not only undo the changes to the latter document, but also navigate back to the first document (e.g., Previous) and undo the changes made there. Since folders are treated like

documents, the same undo model would apply to folder manipulations. I'm assuming a multiple-level undo model such as Word has implemented.

Printing is another area where we can provide a richer framework in the shell. The shell's print dialog should be a subset of the one in Office. The basic principle is that all views should be printable, with fairly reasonable print formatting and a decent set of print options. Specific apps can augment this feature set as desired via frame negotiation.

People in Office have proposed one idea that fits well with the idea of generic New and Print capabilities, as well as generic viewing: they want to make the app/frame hosting protocol be based on exchanging HTML between the app component and the frame. This would make the frame entirely responsible for display rendering and (by logical implication) printing. I think this idea has a lot of potential because it really pushes on the idea of making documents completely HTML-based, plus it makes sure the shell has all the code needed to view any HTLM document.

Another function ripe for shell/app integration is the Office assistant. This should become the Windows assistant. We'd define the conventions and protocols that let apps extend the assistant's knowledge base and behaviors.

To sum up, I'm suggesting here that we take advantage of the great work Office has done in providing a consistent UI across the component apps. What we can do now is use that (or an evolution of it) to provide the guiding framework for the shell frame UI. We put enough features into the shell frame to make the shell the best hypertext viewer on the planet, and let Office and other apps slot in the rest of the features. Of course, because the shell frame design is guided by Office, Office will slot in the best.

Finally, rationalizing the UI this way helps with the frameless UI concept discussed earlier. To make navigating in a single frame among different kinds of information as seamless as possible, the in-place UI must be relatively homogeneous across information types. Using a superset like Office as the starting point for the shell frame will help us achieve that.

6 *The Personal Newsletter*

I return now to the idea that headed up the user scenario in Section 3: the personal newsletter.

All the ideas I've discussed up to now have been about providing a set of UI tools that the shell and apps can take advantage of to provide a seamless way for users to navigate among all their information, and to view and navigate it using a small number of views that are applicable across all that information.

This is all great but it's just tools. What's possibly most important about the hyperactive desktop is the way it exploits push model to go beyond giving the user tools: Its mission is to give the user information, too—the information the user most needs to do their work for the day. The personal newsletter is how this mission is served.

Think of the newsletter as the user's primary launch point. The launch not for apps, but for information and for tasks involving that information. It is the most concrete manifestation of IAYF.

The newsletter is a dynamic document synthesized by a newsletter agent on the user's desktop, based on an authorable template. It creates each periodic "edition" of the newsletter by scanning information sources and alerts based on user profile information, including explicit user subscriptions and implicit subscriptions gleaned from enterprise model information about that user, their position, and their projects.

Section 3 gave a pretty good overview of what's in the newsletter, and for even greater detail and lots of discussion on UI and infrastructure, you can read the "Beyond Browsing" memo.

What I'd like to do here is discuss a few of the more pragmatic options for implementing the newsletter.

The key to the newsletter's usefulness is its ability to present information the user really needs to see, and in an attractive, easy to skim and drill-down form.

For some of the information it covers, this is fairly easy. For example, Hot Mail can show any urgent items first. Your Schedule can just show the next two or three upcoming meetings and reminders. Project Changes can show the two or three most recently changed project documents, and any documents authored by this user that has received new comments.

All these lists of course can be scrolled or linked-to to show more items.

For other kinds of things like company and industry news stories it's a little harder to know what's relevant to this user. In "Beyond Browsing" I discussed a data mining model for how this could be done (and how that would also serve all the other cases I just went through). But let's assume it'll take us a couple releases to get there. What other approaches are there in the meantime?

The obvious one is an explicit subscription model. For each column in the newsletter, the user can fill out a subscription form giving the match and raking criteria that should be used to fill it. While things like mail and schedule can probably be defaulted well enough, certain other categories like news and special

interest will need explicit user subscriptions. (In the case of customizing mail and schedule via rules, you the rules amount to the subscription).

A problem with a subscription model is that users have to decide to perform a subscription action. A way to make this easier, so that it can happen in the context of a user getting information, is to introduce a control in the UI by which users can express their level of interest in a piece of material they're looking at. The Beyond Browsing memo discussed how this would be used in a data mining context. But it can also work for simplifying an explicit subscription process.

For example, suppose each content frame has a small control on the frame border that acts as a linear "interest" scale. You can click anywhere along the scale to indicate your level of interest in what's currently being viewed. The software could then get some attributes about the thing under view—who wrote it, what project is it part of, what kind of document is it (marketing plan, etc.), what are the keywords and other attributes associated with the document. This information could next be checked against your subscription list to see if a new subscription may be needed. If so, the user could be presented with a dialog to add what's needed. This "Add New Subscription" dialog may entail a set of choices so the user can indicate what characteristics their interest was based on.

Project-related information exists in a middle ground where there may be some need for user-supplied customization on what stuff to care about, but where the system can also have a lot of built-in knowledge on how to figure this out. The key to the latter is having an explicit representation for what I call the enterprise model.

The enterprise model is a collection of databases, most of which already exist online at most companies, like the employee database, org chart, and project databases. If this set of information exists in a format that shell and app components have access too, you can see how software could be given a lot of intelligence about what matters to a user.

For example, when a document is changed, attributes of the document could indicate what type of document it is (say, marketing plan versus test plan) and what project it's a part of. Org chart and project databases would then indicate what people care about reviewing this document (based on the project name, document type, and author); and a link to this document could be inserted into their newsletter.

Basically, the enterprise model gives the shell and apps a model for how to connect documents, people, and projects, so that changes in one of these can inform the others. Also, because subscriptions can be associated with any of these objects via the model, it's possible for a person to "inherit" interest in certain news categories, for example, just by being assigned to a project or by

telling the system that you share another person's interests in some category (e.g., news).

The "Beyond Browsing" memo has lots of information about the enterprise model. In summary, it includes:

- Employee database. Provides user name and attribute information.
- Org chart. Relates people to each other and to projects.
- Position database. Provides the name and attributes for each position and job function in the company.
- Project database. Provides the name and attributes for each project in the company, and links to the where project databases and documents are stored. May also express project relationships, like dependencies and contingencies.
- Process database: Same as the project database, but about corporate processes. (Like a project, a process is something a specific set of people are involved in, and which as associated documentation and terms of relevance).
- Email groups database. Provides the name and attributes for each email group, and a group profile derived from the individual member profiles. (The group profile expresses things that are of common interest to group members, useful for directing information to group members.)
- Document templates. Provides standardized templates and meta-templates for various documents like project proposals, product specs, marketing plans, budgeting worksheets, and so on.
- Enterprise dictionary. Provides keyword and concept terms and property names of significance to the enterprise. This dictionary is an accessory to the content dictionary used to do feature extraction of objects.
- Profile database. The set of user profiles, as well as model (synthetic) profiles for various company positions and projects. Model profiles are optional, but highly useful, because they encode knowledge about what kinds of information and types of documents each position and the members of each project are likely to be interested in.

Given this information, the system can now get answers to many questions concerning what a user will be interested, covering such topics as:

- Email: most interested in mail from boss, direct reports, managers up the chain.
- Projects: most interested in projects assigned to, and contingent and dependent projects.
- Documents: most interested in documents for projects of interest, especially document types pertinent to position (e.g., product specs for a development manager), and policy and procedure documents pertinent to position.
- News, articles, and reports: most interested in items matching project terms, and terms related to position.

Beyond providing the "right" information, the newsletter also needs to present the information in a nice way. Partly that says it really should look like a newsletter: a thing with narrow columns, with good use of typography and illustrative graphics and other eye candy to make it nice looking. To make it easy to digest, the "front page" should have digests of the key information being linked to, and not just be a pile of links.

Its format should also be customizable, at two levels: at an authoring level, it should be possible to control the layout and presentation features of the newsletter in detail, and to specify the inclusion of content, content categories, and subscriptions. At a user level, it should be possible to rearrange the visual ordering of the material (the "columns"), to increase or decrease each column's depth of coverage, and to tailor the subscriptions.

Note that the subscription model for the newsletter differs from the traditional one, in that the newsletter acts as a single focus for pulling all subscribed information together into a cohesive presentation. Having multiple subscriptions doesn't imply having multiple disparate places to look or alert streams to monitor.

The newsletter can be more than just an information feed. Because it provides a view on project documents that have changed, it can also be a focal point for collaboration.

One scenario: My newsletter informs me of a change to a project document of interest. I click the provided link to open it and add a few review comments. Whoever wrote this document will get a notice in their newsletter that one of their documents got comments back. They'll click on the supplied link and, based on link settings in the newsletter, the document will automatically open in the Comments view, making it easy for them to review the comments. They can then comment on my comments if they desire (remember, you can add links, and this includes annotations, to anything).

Now, because the comments *I* created now have comments, my newsletter will inform me that I have comments back, and a link will take me to the document, again open in Comments view, and with the comments pane categorized on thread ID (thread view) and filtered for read/unread. This makes it easy for me to focus on only the appropriate comment threads.

7 Applications

The hyperactive desktop changes what it means to be an application. This change is very much in the direction we've been trying to head for several years: that of a document-centric world where apps are components. What the hyperactive desktop does (or at least tries to achieve) is to componentize the UI in a way that makes componentization of apps more possible.

These components are the various elements of the shell frame I've discussed: the various kinds of control and client panes and the features associated with them, and the set of standard, negotiable/extensible UI primitives: open, view, edit, undo, insert, search, next/previous, print, help (user agent), and so on.

Up to now, when we've thought of "document centric" we've thought SDI. But the hyperactive desktop takes this one step further. Not only is the model "single document", it is also "single frame": single frame in the sense that when you navigate via links, the default action is to follow the link within the current frame, rather than create a new frame. This is nothing more than the internet browser model.

In this single-frame world, applications become controls that are hosted inside of HTML documents. You don't run apps, you navigate to documents. The components in view in the document negotiate with the shell frame to add their unique menus, toolbars, and UI forms, and to embellish existing ones. Visually, these changes are usually minor, since the default is for most control panes to be auto-hidden, and for most embellishments to be straightforward additions to the existing structure. For example, adding object types to the Insert menu, browse types to the next/previous type list, and view names to the View menu.

Another attribute of the single-frame model is that there is no longer a difference between an app object being full frame versus an embedded frame (in place OLE object). In either case, what you have is an app object embedded inside the shell frame. The only difference is whether the display region of the app object is a subset of the containing pane versus the entire surface of the pane. The same app/shell frame negotiation occurs in either case.

On the hyperactive desktop, a folder is an app object like any other. So as I've already discussed, you can author them, embed open frames in them, and embed them in other kinds of objects. This uniform treatment of folders, coupled with the rich linking model, is what makes it possible for the shell's folder code to act as a universal client for a wide variety of information types, including documents, mail, newsgroups, appointments, and web pages. Structuring the folder viewer as an app like any other is quite important. Think of it as a prototype that other apps will use as a model for how to integrate into the shell frame.

The final point I want to touch on has to do with the relationship of apps to content. In a document-centric, hypertextual world, there really is no dividing line between app function (UI) and content. Expressing an app's help system as a set of web pages is one obvious example. Put dynamic content on those pages and you are now expressing the Office agent the same way. Indeed, as Eric Michelman proposes, and entire app's functionality can be expressed as a set of navigable web pages. As in: the help and agent pages for an app contain buttons for invoking the operation being explained.

I agree that this is the right direction to go. A web UI has some big advantages. It means that:

- The user can use the familiar navigation, viewing, and searching tools to get at app UI functionality. These tools work for remote as well as local information, so parts of an app's UI or its add-ins can be internet-based.
- An app's UI would be authorable using the same tools and techniques that you use to author any kind of content. The UI could be made just as attractive as any set of web pages.
- The app's UI could be customized by IT people or even end users to any degree the app writer wants to permit, using normal editing tools.
- Apps could be upgraded incrementally and transparently, just by modifying a subset of their pages.
- UI clutter can be further reduced. Standard menus and toolbars need only carry the most essential features. Users can find others app functions by navigation, search, and agent suggestion, and they can then drop any they use a lot on control panes of their choosing.

A key to making the web UI work is for the app author to organize the UI pages the right way (much as the author of any long document has to organize its topics correctly). The author also needs to make sure the pages carry the right kinds of links and search keywords to facilitate discovery.

For example, the app author should make use of some of the organizational kinds of links I discussed in Section 5.5.2, like Topic links. We should add to this list of types as necessary so that we have a standard set of link types that are useful for conveying the semantic organization of a complex body of material. This set of types then lets us embody a common set of views and previous/next behavior that other apps writers as well as general content authors can use.

What I'm saying here is that today's web linking model is insufficient. It lets you link things together, but with no description about how the material being linked is related. To enable intelligent navigation and viewing features, we need to encode more semantic information into the links, and apps (for one) should take advantage of this in organizing their UI pages.

Content can also contain meta-information that implicitly relates it to one or more classification hierarchies; this makes it possible to perform intelligent search, navigation, and viewing of information in the absence of explicit links. This is important because new information can be introduced into the world more easily if its relationship to other material is implicit in its meta-information—versus needing all such relationships to be explicitly coded as links. (The latter is more than just labor intensive; it presents a combinatorial problem and implies maintenance effort when information changes.)

Another thing meta-information can do is provide overview information about the information being described. Newsletter agents would use this to create the "headline" and "lead text" for personal newsletter entries, thus making it possible for the author rather than a robot to generate the lead material. The shell viewer would also use the overview information when displaying link lookaheads. Note that this information can be graphical as well as textual—it is HTML.

As a simple example of meta-info in the app domain, apps should imbue their content pages with search keywords and properties designed to assist users in discovering app features. Thus, all print-related functions would have a "print" keyword, and all features of an app that are considered to be "advanced" features would have an "advanced" keyword. To discover all advanced printing features you could search on the AND of those keywords, or you could navigate to a built-in printing features page and filter on "advanced". Naturally, apps could predefine a bunch of the most useful UI-discovery/navigation views and include these as standard pages in the app's set of UI pages.

I'm sure there are a set of standard search keywords and views we could define that would be generally useful across a wide range of applications, and this would be a useful standard to set. We should also pursue work on a more general meta-content language to enable intelligent searching, navigation, and view synthesis.

8 *Next Steps*

While this memo has covered a lot of ground, for the most part I believe it represents thinking that is evolutionary and that builds on things we've done so far, or known we want to do..

The desktop model is a generalization of the Memphis model and from a UI perspective is backwards compatible with it (mainly it is a superset). It should be straightforward to flesh out the UI details of what I'm proposing.

The new shell frame and its various parts—like navigation and tracking panes and the customizable menus and toolbars—are mostly generalizations of existing features we have implemented in various places, including the Memphis shell and Office. Likewise, the Notes-like searching features are just transplants of features

we've already implemented in Outlook and Exchange, or else have wanted to. Fleshing out UI details for all these things should be straightforward.

The biggest design steps I've proposed here concern how we can

1) Present a single model for folders and documents
2) Exploit rich linking to unify a variety of operations and to support collaboration
3) Use a "single frame" UI discipline to reduce UI clutter, achieve a more web-like model, unify the treatment of windows and in place frame, and enable the "web UI" model.

The single document/folder model is of course the conceptual basis for our existing efforts to provide a web-centric desktop. This is what web view is all about, at least as a baby step. The next step is to more fully realize the model, as I've outlined in this memo.

The rich linking features are for the most part just a matter of making the detail UI decisions and writing the code. The biggest UI challenge here is handling external versus internal links—how well can we keep the user from needing to make choices about what type of link to use. I think we can solve this pretty well by keying the choice off of the specific task the user is performing.

Probably the biggest technical risk concerns the single frame model. A lot here hangs on our ability to define the right UI framework so that a full-bore frame negotiation model can really work. I think we can do this, because it's an extension of things we already do in Wordmail and OLE in place editing.

The idea of control pane hiding at a window level is something we'll need to prove out through experimentation. This is a major element of how I'm suggesting we cure UI clutter, but it has no bearing on any of the other UI features I've discussed.

Another challenge to the shell frame UI model is Docfile performance. Since the single frame model depends heavily on frame negotiation, and the most logical starting point for this is Docfile, we need to solve the Docfile performance problems or come up with an alternative design.

Finally, to truly exploit a web UI model for app features, much work needs to be done in reorganizing apps as a set of linked pages that host UI controls. To facilitate this, we need to define the conventions for links, keywords, and other meta-information that will be the basis for navigation, searching, and viewing the app UI pages.