Memo: **What is a Document?**
To: Bill Gates
Cc: list
From: Greg Whitten
Date: October 26, 1993

How should the question be answered?

I decided that there were two useful ways to answer the question.

1. Define an ideal (or model) solution, i.e., the objects and relationships.

2. Outline what we could do to our existing products to move us towards a more ideal solution by showing how the changes start to satisfy the requirements from 1 above.

Most of what I have written in my various memos already addresses the first way above by covering requirements and areas of design that need to be done.

What I want to do now is to explain a little about the user model surrounding documents and how I think that we could evolve our existing products. I think that the second part of this response will be more concrete and more valuable to understand over the nearer term, hopefully, clarifying some of the development cost vs. user benefit questions. My priorities for specific items will probably not be the same as yours; however, I will try to order the items so that the changes which are dependent on others will appear later (topologically sorted).

I can always address the ideal later, as needed; but, hopefully, we should be creating the ideal by following the proposed tasks.

The quick answer

I also have a third useful very abstract answer that characterizes the scope of the problem to be solved - a document is a container of organized information.

Key Perspectives

There are three key perspectives to this question that should be kept and addressed in the analysis of this problem - What is a document?

1. The key implementors of Microsoft documents - integrated office and desktop applications. What and how much can they do? Compatibility? Competitive features? Performance?

2. The key software clients of Microsoft documents - Ren, Cairo, and work group applications. Documents are interesting. What can I do with them? How can I find information in them or about them? What is the externally visible structure that I can write tools to?

3. The end-users who use, customize, and build solutions using Microsoft documents.

There are other important aspects, as well.

4. The information that documents contain, organize, and manage. This is important to information creators. We should try to make our information types richer and more flexible.

This memo is incomplete and I am not going to finish it !!

I am not going to continue and try enumerate everything. I want to keep my part of the message simple and direct. Instead, I think that the collective WE should try to construct or contribute to a framework of tasks that when completed will let us build the software that we want our customers to have to solve their problems. Our interdependencies need to be collectively understood and exposed.

If I have not answered enough of the question for you, please ask more specific questions and I will try to give relevant feedback on the problem. I may have an answer, an approach for tackling the problem that will lead to a correct solution, a set of requirements that a good solution to the problem should satisfy, or an I don't know, haven't thought about it.

I probably will not design or implement any part of these products. I am only suggesting a consistent way that I would use to solve the problem of how and what to design and implement.

Lastly, I am always looking for alternatives, I may not be right. I only care about delivering the best we can practically do in ways which do not seriously hamper our future product progress. I also believe in first things first or getting the kernel requirements satisfied first. Aligning a group requires a collective vision, road map, and pragmatism including compromise.

High Level Product Improvement Goals and Competitive Environment

I think that it is important to have a few product goals when proposing new work to be done. Here is the list that I am using for my task analysis and breakdown.

List of integrated office product goals.

- Improved ease of use
- Better integration
- Better programmability
- Working with other Microsoft software better
- Reduction of product and development redundancy and complexity
- New document directions
- Improved features
- Improving Integrated Office's objects
- Examples for new functionality using the improved objects

The competitive environment is such that office suite sales are going to dominate for a period until best of breed components become significant differentiaters again. The shift back can really only occur as stable architectures are developed for components. Componentized reusable software is the solution that matches many of the above goals. Our efficiency as a development organization may depend on this as the only way to control the complexity of huge projects. If we do this work, it will make it much easier for us to keep our customers because of their increased dependence on our architectures. Changing products will no longer always be a matter of data format conversion.

MS-PCA 1431785

CONFIDENTIAL

MX 6045930
CONFIDENTIAL

Creating Integrated Office Documents from Office Documents

This section contains tasks that take us from today's documents to tomorrow's documents which satisfy the product goals. The task recommendations need to be considered carefully because they need to satisfy the goals and the requirements for the three key perspectives. Thus, some tasks will address the internal implementation of an Integrated Office document and other tasks will address the external interfaces for the document abstraction that the client software tools require.

Creating a Problem-Solving Mindset using OLE 2.0

The problem needs to be solved using OLE 2.0 compound documents as the base. Microsoft has just made a large investment surrounding OLE 2.0 and Integrated Office provides a good opportunity to build on that investment.

There is a basic level of understanding of OLE 2.0 in the product groups. We will need to go beyond that in answering the question - "What is a document?"

The first step is to ask ourselves is "How could we have done a better job supporting OLE 2.0 in the products (in this case - Office)?" I am not going to answer that here. The second step is to believe that the problems can be solved using OLE 2.0 where possible and using extensions to OLE 2.0 for new requirements. The most widely used extensions will become the kernel of the new OLE 3.0 design work. If we have problems with OLE 2.0 or 3.0, then we try to work together to fix them successfully.

This mindset needs to be growing in development, program management, testing, and user education. (Marketing is probably already selling it in the current products!!)

Understanding OLE 2.0 and Compound Documents

I think that my abstract definition of a document as a container of organized information can help one understand the most important concept of OLE 2.0 compound documents and how we can move beyond it by decomposing the problem a step or two further.

An OLE 2.0 container contains OLE 2.0 server views. If the container is a document and the view is a view of information, then an OLE 2.0 document is a container of views of information. The OLE 2.0 container implementations provide the organization features for the information.

OLE 2.0 provides a user model and user interface standards for containers and objects. The container aspects of the model come from the familiar selection model including drill-down activation, direct manipulation including drag-drop, and commands for complex tasks including persistence and object instance creation. The object orientation is expressed by the commands which operate on the object including context menus, property sheets, and direct manipulation of parts of the view. The container aspect of an object is also supported with parts of a view. The user model includes the following two expectations - 1) if I have a view, then I can get to the object and 2) if I have a object, then I can create a view. Lastly, OLE 2.0 offers a degree of transparency with respect to the question of embedding or linking (sharing).

One of the decomposition steps that I am proposing will let us separate the view of information object more explicitly into two objects - view and information. The benefits will be apparent in several of the following tasks.

Lastly, the container and view of information split present in OLE 2.0 can let us consider what a product would look like if the "best of breed" components were selected.

MS-PCA 1431786

CONFIDENTIAL

MX 6045931
CONFIDENTIAL

User Model for Information, Documents, Workbooks, and Workspaces

I think that it is important to have a user model that can clearly encompass and distinguish these types of user level objects. We should be very careful when we start to create product features which needlessly blur the distinction, i.e., just because it is possible to do anything in software that does not mean we should do it. Certainly, by having a clear, consistent, and parsimonious user model for our software we will be improving the learnability and usability. Integrated Office is the ideal opportunity to make the leap forward.

For the purpose of this discussion I am equating information with the OLE 2.0 compound document server views which should include the selectable parts of documents. The problem that remains is how to characterize and differentiate documents, workbooks, and workspaces. The discussion will stay at a fairly abstract level until I start discussing actual implementations.

The remainder of this part of the discussion is necessarily abstract because we are considering three types of user level objects which fit the same abstraction - containers of organized information.

Documents

Documents only have value if the contained information is accessible. There are three primary forms of access to the information - programming, viewing, and printing. It is NOT necessary for all documents to support all three forms of access. However, the information objects should because they could be in any type of document . Our generic document implementations need to support all three. A document instance contains all the state necessary to control its programming, viewing, and printing. I will discuss a way to structure the implementation work for this later.

Today's documents have very limited ways of organizing the contained information. For example, Word is almost completely oriented towards sequential layout of information with designated paragraph styles providing the hierarchical outline structure. Word's organization structures are not suitable for DTP style documents. However, the component functionality of Word is close to what is required. It is this breakdown of function into reusable components and well-structured relationships that will give us a lot of product flexibility in the future. I think that an easy way to see the possibility is to consider two Wizards - one for WP documents and one for DTP documents. These Wizards to do not actually build the document, instead they configure the structure of the implementation by selecting different document organization and layout components. If you can start to do this, the flexibility of the component set can be leveraged very quickly into new products or functionality that can provide value for users.

I believe that it is important to be able to organize information in many different structures. WP and DTP are the obvious choices with on-line documents and SGML (or similarly structured documents) being the next opportunities. Conversion between organization types and potentially different policies for display and print should also be considered.

I wrote a section about componentized documents in my APPA Mission and Notes memo. BobAtk also wrote a paper about documents that is relevant to the discussion.

Workbooks

Our designs for workbooks simply treat them as a linearly organized container (including storage) of heterogeneous documents and information objects. Workbooks are also documents in their own right which would imply that they should have their state for organizing, viewing, and printing. Since workbooks can also contain whole documents, it is easy to see how this kind of design can create a lot of dissonance in organizing, viewing, and printing due to the fact that the workbook state and the sub-documents state could be completely unrelated. Non-document information objects might get some of their viewing and printing state from the workbook.

CONFIDENTIAL MS-PCA 1431787

MX 6045932
CONFIDENTIAL

Consider the following scenario that illustrates some of the problems to which I am referring. I am writing a document that contains a few charts and tables that I have already created. I decide to use a workbook. I create the new WP document in the workbook and add the charts and tables into the workbook. The workbook is useful while I am modifying the objects and flipping between them. Occasionally, I think it would be nice to see two of the tables at once to compare some information, this is no longer possible since I moved the tables into the workbook. Next, I decide that I want to incorporate the tables and charts into my WP document. The way that I do this is to use OLE 2.0 to create links to the objects. If I decide to embed the tables and charts in the document, then I should probably delete them from the workbook; however, then I lose my ability to access them with the convenient tabs. After doing a lot of work I decide that I want to print my work out. What should I print out - the workbook or the sub-document? Suppose while I was creating the document that I created some workbook printing state because I wanted to print out everything in the workbook especially those non-document information tables and charts. Where am I provided with the choice in the user interface? Suppose the workbook actually contained a few more documents including one that was set up to be faxed and some other tables and charts. What would have happened if I embedded everything and just was left with a workbook that contained a single document with the two sets of printing states? Suppose that the workbook was designed to show either its table of contents or one of the contained documents or information objects. Can the user interface really provide the intelligence to make the right choice when I push the print toolbar button? What is the algorithm that has to be used to select among all the choices that might seem reasonable given all the different pieces of state?

I picked the above scenario to highlight some questions that we need to answer about workbooks. They seem quite reasonable and usable until entire parts of a workbook are operated on or the document in a document dilemma is hit. How other products deal with this problem should be explored. However, we have to be careful in our analysis since some products which use workbooks heavily only have ONE workbook and all document manipulation is done through that simplifying the actions and user model to one where the parts are really all documents and the whole workbook is never operated on. In our system we will have many workbooks.

In the user model workbooks have also been distinguished by the use of a set of linear tabs to switch views to different pieces of the workbook. How is the order of the tabs determined? There are many useful possibilities - sequential, alphabetical, hierarchical, favorite, most relevant, MRU, the ten places in the sub-documents that I want to look, etc. The best answer might depend on a user mode. How would a user pick one of the many possibilities causing different pieces of code to be executed to provide the different lists?

Consider the following scenario for the tab UI. I annotate a document that is saved in a document library. I create a list for my annotations in the document, turn the most important ones into a tab set, and send this tab set to someone to use to look at document and conveniently find all of my most important annotations. In the past all I could do was to mail a reference to a document, now I want to mail the action -view a document with a given tab set. I have created a scenario for the use of tabs which applies to documents instead of workbooks and has little to do with the document's structure.

This above type of scenario can be generalized to other information structuring viewers, devices, and algorithms. In the case of tab views the questions for us as implementors would become the following. Do documents have tab views? What structures do I need to see as lists and where do I place that list view in my UI. How do I choose the list to be viewed as tabs? This implies the need for a favorite list of potential lists that are initially available for a generic document. Lastly, how do end-users create their own lists?

My conclusions about workbooks that contain documents could be summarized as follows - workbooks are not good documents by the same measures that we would use for our standard Integrated Office documents, workbooks are more understandable with semantics we would associate with folders, and tabs are good viewing organizers for documents and other containers especially if they can be customized and a single view at a time is acceptable.

I think that workbooks are a useful container in the user model. At this time I have a rough design in my head that I think answers the questions that I have raised about workbooks and keeps them properly differentiated from the standard Integrated Office documents. It can be close to what has been designed to date with some more well defined semantics. I will outline this and discuss it with the Integrated Office team. They are responsible for the final design.

Folders, Desktop, Workspaces, and Tasks

I was originally going to address only workspaces in this discussion. I added a few more user level objects so that a hypothetical relationship between them could get discussed.

In Win 3.1 only the data part of a folder exists as a file system directory. The viewers for directories only exist in the File Manager and in dialogs like File New/Save As/etc. Files stored in a file directory do not really have any independent behavior, instead the file manager implements all of the "object" activation policy using associations or determining that the file is an executable from its filename extension. (In Program Manager the activation behavior is determined by the state of the item in a group. Packager implements similar semantics as a truer OLE object.) A simple abstraction for a folder would be just to consider it as a collector for objects. High level operations would exist to move, copy, delete, and activate the objects contained in a folder. The desktop as a data entity should also be folder-like with a different viewing metaphor or user model. Our next UI designs adequately recognize these requirements.

Workspaces and tasks are new concepts in the Windows user model. I am proposing a hypothetical set of requirements that a user or system builder could depend on which also establishes these as entities in the user model.[1]

A workspace is also a container or collector of objects; however, I want to distinguish it from the desktop and folders. The single top level requirement that I want to add is that a workspace can have well-defined behavior associated with it. The standard desktop is can also be a workspace, but the user is allowed to make ad-hoc changes to its behavior as he (or we as implementors) sees fit. Tasks are also able to be well-defined entities that can "live in" or work with well-defined workspaces. By allowing some part of the users environment to have a more rigorous definition we can enable businesses to implement mission critical applications on the desktop without fear of the potentially ad-hoc nature of the users desktop workspace and application customization. By keeping this top level robustness requirement in mind when we actually design the various components it will make some of the decisions less controversial. This robustness requirement should also help us when designing new customization features for our other office objects since they will have to coexist with workspaces and tasks built out of the same components.

The primary ways that robustness can be enforced in our environments is to treat encapsulation more seriously, keep user, desktop, and workspace customization in independent instances, provide for a better separation of contexts so that the appropriate customization instances can be selected, and to consider some degrees of flexibility that will still permit a user to work with tools (editors and viewers) more to his liking on the underlying data. Cleaning up our object customization and add-in models is a major part of this work. Further scenario and requirement development is necessary to make this into a completely compelling argument; however, I think that with very simple scenarios it is easy to see how our current products break down. I think that it is time for rethinking and not just repair on this subject.

Work group applications are the biggest beneficiaries of this type of robustness. This is perhaps something that the Lotus Notes environment can not provide. A challenge for us to figure out is how to sell this as an important feature of our software set. What are the dynamics that would make each class of buyer say that they need this type of behavior? Work group, perhaps, almost lives and dies by this; however, they can not make the desktop

[1] I have purposely left workspaces and tasks as vague or abstract objects. We have ongoing design work that maps to these conceptual objects that I have not reviewed. My major concern in this part of the discussion is related to robustness. We need a user model that can support mission critical applications on our desktops. We need our applications to be more than front-ends for remote databases or data engines. They need to be components in distributed peer-to-peer systems that can run on desktop machines.

application teams see this as an important requirement. This was also a frequent message in many of my memos about application architectural issues.

User Model Summary

What ever Integrated Office supplies for workbook-like function, it needs to be consistent with its environments - Win 3.1, Chicago, Ren, and Cairo. Given that the folder views and smart folders under design from these environments are starting to impinge on this part of the user model, there is some serious analysis left to be done including rationalizing 4 teams development plans so that we are not creating so many workbook-like implementations that are so inconsistent with each other that we can not create a user model. The instability of the various shell designs today has to add to the confusion especially when 1) the actual requirements of any new user model have not been well specified by any of the teams, 2) the teams do not have a joint strategy including compatible requirements for delivering a user model, and 3) the end to end design process is so weak. The creativity in expressing the UI should come after the analysis of the user model. I think some progress has been made in this area by some of the recent changes by moving towards a single team, but that does not mean that the team is analytical and rigorous in its problem solving and controlling its creativity.

Mapping the Answer to Components

I have put together enough of a high level picture that it is possible to map what we have to a set of components that satisfy what I think some of the most important system-wide requirements are for our software in the future. Rather than doing that here, I will try to show in the following sections how we can decompose and evolve our existing Office products into some of the key components.

Key User Models for Information and Direct Manipulation

One aspect of the above discussion was to identify a few classes of user level objects to which end users could relate and to which we could map our current products. The missing aspect was defining a consistent set of high level operations for those objects. There are a set of basic operations that the user model should include that are generic for most objects - create new, copy, delete, add to container (ala drag-drop semantics), and save. We should also anticipate some of the new generic operations that we want to promote with our next Integrated Office releases like compare, merge, and version. This email is not going to answer questions about these new operations.

I don't think that we need to go beyond what Windows and OLE 2.0 allows for an object oriented user model and interface, we just need to get more consistent in the use of OLE 2.0. I.e., drag-drop, drill-down, context menus, and container-containee UI negotiation should be supported more meaningfully and consistently by our containers. I don't need to provide an analysis of the current state here. Someone should just produce the matrix of direct manipulations and resulting actions for the various contexts in the products including the Integrated Office that we will be producing in the future. Consistency and/or problems should be obvious. It is possible that some additions or negotiations would allow more seamless integration. These should be considered as part of OLE 3.0 or Integrated Office standard extensions to OLE 2.0 Which direction to take may depend upon how generally applied they can be.

I think that drawing as a user model and its associated direct manipulation user interface is the next thing beyond our OLE 2.0 UI work to concentrate on. The model also needs to address multiple layers in additional to Z-ordered objects on a layer. The compelling reason for this is the number of times that we should be using the drawing model to expose the construction and manipulation of objects that the user is going to view on the screen. The some of the places in Integrated Office that we should anticipate using drawing as a model are 1) drawings, 2) page layout, 3) annotation, 4) form construction, 5) charts, 6) construction of composite information elements with constrained layout, and 7) an advanced printing model (see my notes about digital paper for more details).

The UI for drawing can reuse much of what we have already defined for Windows and OLE 2.0. There is one more piece of the user model for which we need to define a set of consistent operations - hierarchical navigation and selection. This can be applied to following objects that are hierarchical - drawings with grouping, outlines, page layout, equations, charts, composites, etc. Mouse and keyboard interfaces need to be defined and used consistently. Since hierarchical navigation and selection can be applied to more types of objects, it may be reasonable to look at defining more friendly or accessible keystrokes or mouse operations.

I guess by now it is pretty obvious that I don't care that much about the user interface of our products other than the fact that the UI elements are learnable and generally applied where it makes sense. I like the "Keep It Simple" model before making it complex and special cased. Extra usability can be added by more in-your-face buttons, toolbars, and tabs for operations and tasks; however, the contexts in which they operate correctly has to be understood in the design process. A lot of our UI breaks down when confronted with compound document, heterogeneous object, or multi-level problems. Our usability analysis has to broaden its scope.

Selecting the Key and Best of Breed Components and Features

The last criteria that I want to cover addresses the compatibility and continuity of the software that we ship. Integrated Office will be a new product, but it has to bring its old customers and their information along.

We need to have a mind set when designing Integrated Office that we are striving for best of breed components. Our selection process for what to keep and to invest in and what to leave behind as legacy. This will mean looking across the products that we have to find the best starting point for moving forward. In some cases the best starting point (possibly largest code base) will not have all the necessary features. Our product plans must address how to bring the best features into the components.

I think that we should also have a two release mind set when designing the component set. This can help cement the team vision for understanding the impact of future decisions on the upcoming product release. It also gives the teams more competitive flexibility during the development process. Usually individual tasks take more or less time than scheduled. This can make it easier to add or delete functionality as the time permits. This is very close to our project decision making today except that now delayed functionality needs to be anticipated in the design work instead of dismissed. Lastly, the functionality of the set of components needs to be coherent. This means that it is necessary to understand the interrelationship of functionality with the various components. Certain development tasks will require equivalent support in other components. Unlike feature teams in today's products, it is not as easy to drop or add a feature with component software. This has to be clearly understood in the abstract and in the concrete as it applies to each part of the product.

This is the end of the abstract part of the discussion. The above mostly represents concepts that need to be incorporated in one's "belief" system that form the abstract top level requirements of any solution. I have found that these requirements are almost never written down when a design is being done and that misunderstandings at this level lead to a noticeable percentage of design disputes.

Divide and Conquer Approach to Integrated Office Components

The discussion in this section is about how to divide up our existing products so that they can be reengineered into Integrated Office components. There are many ways the entire problem can be solved including ground-up development. My recommendation is to take a more incremental approach to the problem that still can lead to a finer granularity component solution. We can take advantage of two things that we understand about OLE 2.0. First, we can always virtualize objects of any granularity out of our monolithic implementations. Second, full componentization is not required given the first. I.e., we can choose to componentize to the degree that WE, as implementors, need to deliver the required functionality. Lastly, we want to be moving towards single implementations or shared components where it makes sense. All of the above leads to a divide and conquer approach to components. We may even find that when we understand how to divide something up that we now know how to build the same functionality from ground-up components. I don't think that we should have to rely on that level of understanding for our next few product releases.

I have discussed the following type of approach with ChrisGr and others over the past year. I think that as the approach is more understood it will be easier to understand the requirements of an organization to produce a more componentized Integrated Office product. This would include the types of design and implementation problems that the various groups will face, who is involved in the solution process, and how to solve the problems and resolve differences. I have written email and memos about this. I can supply copies as required or requested.

Where is the obvious place to start dividing? The answer is at the OLE 2.0 compound document interface and user model level. This lets us look at our products as compound document containers and compound document information servers.

Our major information types - text, tables, charts, drawings, databases are all sufficiently different that there should be little argument about them be able to share the same implementation(s). For now we will assume that they are separate because if we ran out of development time that may reflect the lower priority of componentizing this part of the problem.

That leaves the other side of the problem to consider - the container and organizer of information. If we asked the question what is our best of breed implementation (i.e., the best that we have today) for this set of function, the answer would have to be contained in the monolithic Word 6.0 implementation. It has the best document layout and printing model and is an adequate starting point for much of the other functionality including storage, viewing, and outlining.

Lastly, the divide and conquer process can be recursive producing ever finer components. I have organized the tasks into phases that I think when completed produce a good level of consistency. The phased approach also creates a more concrete set of component software engineering goals at each step. I think that this will prove to be valuable for the design and development teams by allowing them to grow into an understanding of component software development issues.

Splitting Word

The document implementation in Word 6.0 is going to provide the cornerstone for the component work by providing the primary document type that is used by Integrated Office. More concisely, I mean the following - 1) when information is viewed as part of a document, the Word components will be the viewer, 2) when information is printed, the Word components will be used to print the information as part of a Word based document. I.e., when an Excel spreadsheet is printed, it is rendered as part of a temporary Word document that manages the layout of the Excel information. Exactly how is this going to be done? Some design work is required, but it is possible. If we solve the problem for Excel, then we have solved the problem for every other information type. If this is not clear enough for the purposes of this discussion, please let me know and I can expand on this. If people agree with this approach, then talk to me for sure. I can outline how the temporary documents and templates should be created and how the object to be rendered should be treated as a link in that temporary document.

Phase 1 - Container

OLE 2.0 assessment - Perform an assessment of Word 6.0 as an OLE 2.0 compound document container. Define where OLE 2.0 can be improved to provide better integration across a variety of data types (try to generalize, but concrete scenarios with Excel are valuable). Some of this design work can form the basis of OLE 3.0. Define where Word 6.0 should have done a better job with its OLE 2.0 support. Outline the design and development work to resolve this and determine conflicts or intersection with work below.

Viewing, printing, and editing - We need to start assuming that Integrated Office documents will be used differently from today's documents. The order of usage in these three areas will completely switch around. Our usability choices in the product should start to reflect this. If Integrated Office documents are going to be used for mail messages and forms, then we have an immediate case for this. The changes that I think we need to design affect how the user sees a document when it is first viewed. Word 6.0 puts up too much of the editing environment interface when a document is opened. This is overkill for reading an email message and it is slow. In-place editability does not need to be disabled, but the user should take an action to bring up the more complete environment (it could a "menu" command or a user preference from a configuration subject to the robustness and encapsulation issues that I think we should address). The editing environment could be brought up for certain document creation and editing scenarios (i.e., user same as author, in-progress or checked-out from a document library, etc.). This should be considered in the context of the overall user model that I alluded to in the first half of the document. Changes here need to anticipate the later phases - see Phase 2 - Container: Viewing.

Page view and paper model - Word 6.0 has most of the implementation done for a multilayered page model. Certainly, layered redrawing works properly, is fast, and handles objects of different types including rich text. The current model seems to have three layers with z-ordered drawing- master page, document, and overlay annotation and graphics. By adding additional bottom layers we could handle what I referred to as digital paper and the printing model improvements in my "APPA Mission and Notes" memo. By having this split we could potentially simplify features like printing mailing labels by treating each label as a logical page and being able to identify which labels on a physical page were still available for printing. The lowest level could be incorporated as a system feature or kept as a unique feature of our integrated office and application software. The same drawing model should be used to manipulate objects and text frames on each of the layers. Ideally, we should be able to use this as a component for multilayered drawings that might fit into frames in the paper model with the same user interaction model and drawing tools. We may need to add an understandable interface for enumerating and switching layers and showing frames used for positioning and sizing objects. Features like headers and footers in Word 2.0 would now simply construct text frames on the appropriate layer. Editing would be done in page view in the appropriate frame selected by the user. We should also consider allowing certain sizing operations to be specified with constraints so that automatic resizing could be done for changes in logical page size, etc. This is not the only place that constraints are useful. Some of the above could be delayed to a later phase, but we should anticipate these features in any implementation work that we do in the first phase.

DTP layout features - Word 6.0 has made a terrific start at this by providing much of the implementation that just needs to be reengineered into a set of components for Integrated Office. We should make an assessment of product deficiencies relative to other products because the above componentization should make it easier to add layout features.

Annotation - Word 6.0 is fairly powerful today. Some types of annotations should be exploit the drawing layer model including anchoring in a lower layer (usually the document layer). Unexpected layout overlapping could be logged as discussed below. I have not reviewed Word 6.0 closely so I don't know the exact function of the product. DarrylR wrote a memo on annotations that had a number of scenarios and suggestions. This should be reviewed again.

Customization and add-in model - The basic idea is that customizations for one document (or class) do not affect another document, i.e., customization and add-in state is separately maintained. I don't want to go into additional detail here since I have covered some of it in earlier parts of this document and extensively in my other memos. It is time to get serious about the problems and not just patching over it as we have done in the past. Without this robustness work it is almost impossible to depend on using our component set in solutions. Most of the implementation work is very easy to do if done in a disciplined way. The feature team model can be used to clean up and do the componentization required for this task.

Document as a form - The WP document with its stream layout model should be able to used as a form in the same manner as a VB form with its drawing layout or an Excel spreadsheet with its tabular layout. The Integrated Office document would need to support the external form interface that is used by the other "forms". The document object needs to be customizable using VB container-containee programming model. I can provide further design directions and requirements for this. The VC team is working on the detailed integration interfaces.

Container object model - The object model for the container that we design needs to reflect the above components. It should also provide view and data separation, generalized selection that supports multiple and hierarchical objects. It should be designed so that we have some future flexibility for new layout and organizing implementations.

Container as OLE 2.0 server - We need to define the set of container level views that we want to support as an OLE 2.0 server. The list should include scalable page views and parts of a document view. Depending on how much separation of the text code from the container that we achieve we could add the following - text range as displayed and text range as text.

Error logging - The Integrated Office document has to maintain its consistency in the face of potential errors. When an algorithmically unresolvable situation arises that forces clipping or truncating information, some user notification should be possible. Ideally, this would be created as a possibly non-printing section of a document. Errors and warnings that are primarily layout in nature would be displayed here and linked back to the pertinent parts of the document. A very powerful way to implement this logging would be to use the annotation facilities in the document. Many of the errors have several easy corrective alternative actions once the user indicates his preference. A form of change annotation could be used for this.

Templates and document layout wizards - The user model should be able to handle the concept of templates which are document generators. Templates could be customized instances that are cloned or procedures (Wizards) which generate the document. There are a number of alternatives for doing this. We should concentrate on two or three that give us the flexibility to do what any of the others would do. We can also have view and data separation in the design of our Wizards which lets the generators to be called with parameters instead of relying on user interaction with the Wizard input forms. With the above functionality it should very easy to create Wizards written in VBA that drive the product including the DTP style documents created by Publisher.

Publisher and other products - It will be expensive for multiple products to make the same investments in document infrastructure. Instead, we can start to consider reusing the components and perhaps having subsets of functionality that are understandable and still use our much of the user model level training that we should be propogating across our product family.

View and data separation user model - The user model needs to augmented to support future notions of view and data separation so that we will have the right usability and expected behavior when dealing with sharing at the user level. Today, our solutions are weak do to the fact that OLE 2.0 did not address the problem and pushed the issue onto link objects which maintained a shared view cache. This should be done early and released as part of OLE 3.0 or 4.0 design work.

Phase 2 - Container

Viewing - Word 6.0 has several viewing modes - page view that shows what a document will look like when printed, a normal view that provides a simple linearized layout view of the main information stream in the document with a couple of views for other information streams - footnotes, annotations, etc., - and an outline view which has its own hard-coded rules for determining outline structure and layout. This is an area where we need more flexibility to address the requirements for a wider range of document usage all the way up to multi-document help systems with webs of information links. The document viewer needs to be able to have more views onto a document that can reflect the various ways the information is structured to and by the creator and to the user (someone who does not care about the structured views a document creator may want). Particular viewers like the outline viewer should become more flexible and able to attach themselves to any hierarchical view of information in a document. Changes like this would let the components be able to adapt themselves to new document structuring technologies like SGML. Things that are parts of our documents today with their hard-coded rules like table of contents, list of figures, bibliographic references, footnotes, and annotations could themselves become viewers. Further decomposition of these would let us reuse the current view implementations like we have today.

We get what we have today, but with a much more flexible architecture that we can use ourselves for implementing new features and other document customizers can use. I can talk about this further. The solution requires some multilevel object / state modeling to make this work. Some of the end-user level querying will use this part of the design. Some of the new design work that has been done with tabs for views should be reevaluated against the new requirements for the user model (some of my earlier comments on tabs are relevant to this)..

View and data separation - The phase 1 part of this was to develop an understandable user model. This task is where we actually change our code so that the document data can be manipulated independently from the view. As we get more client uses for the information contained in documents, we will need to do this to satisfy the performance demands of these clients.

Frame and information composites - The basic idea for this is that interesting composite view and information structures can be created and treated as a unit, i.e., labeled figures and pictures and titled stories for DTP. We could supply some useful ones with the product. It should also be possible for a user to create one using the above component set with the hierarchical layout and constraints. A wizard could help with the construction of a composite template. I have written about this in my other documents. This is more powerful than simply having a wizard which does the construction producing a set of objects that can not be treated as a whole. Simple grouping would satisfy much of this; however, the treatment of constraints by grouping operation may be a tricky problem.

Phase 1 - Word Information Types

Text - This corresponds to the text processing parts of Word 6.0 including the piece tables. The goal of this task is create more independence of this part of the code base from the container. Also, since the text stream can handle other objects, the text stream implementation should be changed to be a container for rich text, rich text objects, and objects.

Text as a control - The text object should be able to function as an OLE 2.0 control. This means that it should respond to ambient properties and be able to generate events when used as a control. The semantics and the event set appropriate for a rich text control need to get designed. VB's stand-alone rich text control can provide a starting point. Ideally, these should be as similar as possible to the programmer.

Text composites - We should consider generalizing fields to support rich text. Some of the current field updating behavior is troublesome, i.e., editing the returned text from a field evaluation and then performing an update fields causes the edits to be deleted silently (which text did the user really want). This is should be corrected by adding read-only fields, possibly as a field property, error logging for the ambiguous cases, and an operation to retain the text while deleting the underlying field. Fields should be reimplemented to this new interface. There are many open design issues with doing this - how does the user choose which field type, etc.

Customizable text composites - We should also provide a VB customizable field class that can supply an arbitrary implementation for an evaluation method that is passed a document context (what context to pass is an interesting question - the field class could implement a variety of choices - a minimum might be an object reference for its place in the document).

Tables - Tables should be treated as objects by the document layout engine. The original implementation of tables in Word was the antithesis of this. If the Word 6 code base did not correct this, then it is time to have tables behave like objects which would clean up the code. I have written about the importance of Word's style of tables before and how some of the hierarchical layout requirements go beyond the flat table views used by spreadsheets. Cells in tables should contain anything that can fit in a frame anywhere else in the product - rich text and objects including nested tables. Ideally, the table implementation that we provide should be a viewer for data tables that support a standard table interface (the implementation split into view and data components could be delayed to Phase 2 - the programming or object model which has the separation should not be delayed). Editability of the data from this view may require drilling down to an actual editor for the underlying data. SGML models for tables should be examined to see if we can support SGML rendering into our hierarchical tables.

Phase 1 - Standalone Information Types

Standalone text - This should have the same data semantics (interface) as our WP text object which can only exist as part of a document. Persistence is not required, but a good idea especially if we use this object for the implementation other components. Primarily used for programmatically constructing text where the view is unimportant. This is the data type that should replace the plain text processing done with strings in BASIC. There should be a rich text viewer which can work with this interface.

Standalone tables - The issues are basically the same as the above. They could also be useful for snapshotting results from queries, filters, or computational transformations. There is some related DAO 3.0 design work in progress for standardizing data access; however, their focus is more on abstracting the capabilities of our existing database engines and not providing this higher level abstraction.

Splitting Excel

The bulk of the functionality in Excel is centered around the spreadsheet as the information object. The container functionality in Excel is simple when compared to Word. The value of the extra Word container functionality is apparent when trying to deal with printed compound documents. Replicating this Word functionality in Excel would be expensive. The alternative would be to be able to use the Integrated Office document for the container side of the product and to shift the investment to the information object parts of the product.

In the task breakdown below I have switched to order to reflect the primary information object nature of Excel. This is a mind set change that the Excel design and development teams would have to recognize and adjust to. I think that the most concrete way to accomplish this is to create a focus on Excel spreadsheets as OLE 2.0 servers for the Integrated Office documents (see below).

Phase 1 - Spreadsheet Object

OLE 2.0 assessment - Perform an assessment of Excel 5.0 as an OLE 2.0 object server. This should be done in conjunction with the same task for Word 6.0 as a container. Define where Excel 5.0 should have done a better job with its OLE 2.0 support. Outline the design and development work to resolve this.

Excel object model - The Excel object model should be reevaluated so that it is consistent and seamlessly integrated into the Integrated Office document object model. This may mean that the Excel application and document definitions will need to change. The object model should also reflect the robustness and customization goals.

View and data separation - This is an area where significant usability in the ad-hoc analytical modeling model has been made at the expensive of this separation. The compound document model which supports shared views will have an impact on Excel object implementation. Ramifications to Excel for this change in the user model need to be identified. They should also be able to handle new data modeling capabilities that I discuss in Phase 2.

Spreadsheet as a form - The spreadsheet should be able to work as a form in the same manner as a VB form or a document. See the Document as a form discussion above for more details.

Spreadsheet as a control - The spreadsheet object needs to work as an OLE control. This means that it should respond to ambient properties and be able to generate events when used as a control. The semantics and the event set appropriate for a spreadsheet need to get designed. This should be an extension of a more generic table control so that these should be as similar as possible to the programmer. This should get defined in conjunction with the standalone table object discussed earlier.

Phase 2 - Spreadsheet Object

View and data separation - I think that being able to have this separation will lead to higher performance of the data engine, higher level modeling, and more powerful data access capabilities for "on-line analytical processing" (OLAP is Codd and Date's name for the area that covers spreadbase and EIS - see PC Week - 9/27/93 p. 113). I think that these directions have to be seen as a big part of the future investment in our technological product improvements. Further details are beyond the scope of this document, but I am willing to discuss a few interesting ideas in this area.

Phase 1 - Container Issues

Workbooks - The big container issue is what to do about Excel's workbook functionality, i.e. how does it get replaced in the user model by our notion of documents and the generalized workbooks. Like many of the objects in our new system workbooks will also need to be customizable. I think that this can be satisfactorily answered by the rough design that I had in mind.

Splitting Powerpoint

Powerpoint represents an interesting challenge since so much of the development work is similar to the Word container, drawing layers, text handling, editable print preview, outlining, master pages, etc. Ideally, if the Integrated Office components could be used to build Powerpoint, then the Powerpoint product development effort could be more fully concentrated into areas where it has unique needs (or where we would want to package the function into a separate product).

Requirements definition exercise - One thing that could be done is to try to define how Powerpoint-like functionality could be created from the Integrated Office components. This can be done by creating objects in Integrated Office that are close to the corresponding Powerpoint objects. The missing functionality can be turned into requirements for the Integrated Office objects or shifted into some Powerpoint components. I think that we might find that much of the container level manipulation capabilities would be useful in DTP-style document manipulations like slide sorting.

Identification of Powerpoint components - Powerpoint clearly has some interesting functionality that be valuable outside of the Powerpoint product. Many of the special coloring, fade, and timing effects could be used in other places to good advantage. How this should be done will require generating interfaces for these components and making them fit into the architecture of the Integrated Office component set.

Cost-benefit analysis - The actual development work for the above to make Powerpoint into a set of components may not be worth the complete cost. Instead, in a situation like this I think it is important to control our investment in the various components using a well-defined multiphase strategy. I can explain in more detail a possible straw man strategy for the above that might provide a good starting point, if people are interested.

Technical Challenges

The above tasks which I think characterize much of the work required to do Integrated Office involve a lot of new design work. There are some additional issues that need to get worked out in the design work that cross a large number of the components. With the new extensibility requirements for Integrated Office these have to have consistent solutions.

Commands, Selections, and Objects

Properly designing the relationship of the selection which encapsulates the set of objects and their hierarchies of selected parent objects to commands which eventually operate on specific objects is critical. The designs for this need to be seamless given that multiple components may be involved. User customized commands need to use this same model in their implementation. This is what will give us the potential for implementing significant parts of our products in VB. A set of multiple object and nested object selection scenarios should be developed to test the design. The design should also be consistent with OLE 2.0 user interface and user model behavior which allows some simplification of the problem to selected inner object and the outer container. I don't have a design or set of requirements for this beyond what I or BobAtk have written since the applications were not ready to work on programmability architectures; however, I can review any concrete proposals.

Views and Data

I think that this is one of the two most difficult and important problems to solve or make progress on. Good solutions can lead to improvements in the user model, usability for sharing and multiple views, and programmability performance. The importance of this separation will also be obvious in the design for generalized selections. We have to stop overlooking this problem. Our implementations may not have to support this as separated view and data components, but they should be able to virtualize this through the programming object model.

Concurrency Model

This is the other hard part of the problem that should get solved with the above design work. The desktop environment is going to need to support more and more concurrency as shared data, external programming, agents, queries, etc. are used. We have to stop overlooking the problem. I would suggest having a long term model for an ideal level of support that is consistent with a more pragmatic solution for the near to medium term.

Data Transfer Model

This is an easy problem to address (and overlook). A single person should look at all the information objects in Integrated Office and tabulate the data transfer semantics between objects. This table can be reviewed for consistency and completeness identifying missing high priority work. Splitting the responsibility for this to the individual object developers would be more likely to miss something.

Programmability

With the Integrated Office work it is time to get serious about standards and architectural issues. I have written extensively about the benefits of doing this and I will not repeat it here. I suggest rereading my memos, extracting an initial set of high level requirements that serve as guidelines, and use them to review the existing object models to point out any problem areas. Properly defining objects and abstractions is difficult and slow work. This effort can be greatly accelerated if the product architecture (component relationships) is really known. The object models should be somewhat consistent with future directions for the product. This will assure the longest possible life for end users' code written to the object model.

Eat your own dog food Challenge

This is a simple challenge to keep in mind when designing Integrated Office document components. The printed manuals, on-line documentation, and help should be able to be produced with the Integrated Office components with perhaps the exception of any multimedia objects (which are just OLE 2.0 objects). I am not trying to say that help should be replaced by these documents, only that it should be possible. In the future on-line (never printed as a whole) documents will become more and more commonplace. Our product designs need to anticipate this and not avoid it.

Summary of VBA Customizable Integrated Office Objects

I have included a list of container objects that should be customizable by VBA.

- Documents
- Spreadsheets
- VB Forms
- Composite View objects
- Composite Text objects
- Composite Table objects
- Custom Monikers and other code fragments including expressions
- Custom queries

Integrated Office System Performance

The current code base for the desktop applications is tuned to a particular use - document creation and editing. In the future the usage of the components will broaden from document creation to include viewing or browsing, querying, and programming against the components in application solutions. The componentization work will have an impact on performance by requiring that interfaces be used in more situations where the old implementations could break the layers of abstraction by directly addressing the underlying data structures. It should be possible with the reengineering work to make some gains back by focusing on the new usage areas and understanding how to make these simpler components perform. The gains at this lower level should be visible to the document creation and editing scenarios. The good news about doing the performance work this way is that the coding changes add to the value of the code base without creating so many new interdependencies that are hard to remember or control as the product evolves. I wrote a talk for a JOOP conference in London that addressed object oriented reengineering. BobAtk and CathyLi both used these as a starting point for additional materials related to object oriented design.

There are some additional issues that should be addressed early in the software design process. These are outlined below. I can discuss further details as required.

- Installation and activation
    - 32-bit improvements - preassigned addresses / fully bound
- Working set modeling
- Separated function to reflect three types of usage - programmability, viewing/printing, and editing
- View and data level activation should reflect the usage
- Multiple instances and code separation - background printing, content filtering
- External querying and navigation

Critical Design Areas, Team Building, and Risk Management

The above componentization work which primarily addresses the top level course grained objects is a new type of work for many of the teams. This needs to be understood and accounting for in the project planning. The project risk will occur at the interface points for the components. This means that earlier design, validation, testability, and stability of these interfaces can lead to quicker development of the key and dependent components. I would suggest rereading my 2-part email memo on "Investing in Architectural Objectives" to help get an understanding of some of the issues as it would apply to the Integrated Office system of objects.

There are some specific process, design, and software engineering issues that can positively impact the plan if the opportunities are recognized early. A few of these are outlined below:

Understanding the generic document goals versus what the clients want
    Properties - implementation required\/ annotational - client / user
Evaluating alternative ways to implement features
Identifying opportunities for reusing solutions
Developing and reusing a new programming model for product extensions
    Use the new document architecture and form invocation model
Getting client requirements for changes they want from applications

Software Clients for Integrated Office Documents

The above discussion is primarily a proposal for the implementors of Integrated Office and secondarily for the end-user perspective. This part of the discussion will cover some of the client software needs with respect to the information in and about documents that are beyond the above.

From an external perspective a document model is only as good as its interfaces. I am not familiar with all the clients (i.e., Ren, Explorer, Navigator, etc.) of documents that we want to provide or even all the implementors beyond Integrated Office (i.e., Powerpoint, workbook, Publisher, Works, Viewer, etc.) of these interfaces. I can not find a visible process at Microsoft for determining our working set of clients and providers which should be involved in the definition or review.

I will try to characterize a few of the external requirements that seem to be out there. Other people should add to the list (remember that I am not going to finish it). My characterizations may not be completely correct in the detail..

Passive and Active Object Models - Whether the object is passive or active is an important point to remember when looking at the requirements that the various pieces of client software. Most of the requirements to date assume that the (document) object is in a passive state. As a result, there is a pervasive model of design which says that a class specific handler is activated to perform the desired function. The implementation for this can be completely independent from the active object implementation. Furthermore, if these passive object interfaces to the handlers were examined, we would probably find that they do not even closely match the object model for the underlying information. Instead, they typically flatten the object conceptually converting it to a single new instance of the type matching the interface. This design approach satisfies the extensibility requirement for heterogeneous information types. Cairo and Chicago have a different way of binding to the required implementations. Cairo uses an aggregation concept in the OLE binding and Chicago must use some non-OLE association in the registry.

(Passive) Content Filters (Cairo) - Information type specific content filter handlers convert the content of an object to a stream that can be indexed by the content indexer. This is a special purpose interface so that it is fast.

(Passive) Preview Filters (Chicago) - These are content view (first page) previews of the object used by something in Chicago (file find dialog and shell?). The views are created without invoking the actual application. The operating system does not cache these views. They are created on demand. This is a special purpose interface so that it is fast.

(Passive) Document Properties (everybody) - I am not certain that the same interfaces are used by everybody. If DAO 3.0 is accepted and implemented by the various data stores, then this problem is ameliorated for most clients. The basic requirement is that an object expose a certain set of properties externally from itself. Each of the stores has a different way of dealing with theses properties. In EMS and Ren the document level properties are moved into a fields in a database row. In Cairo the properties are available through both the top level storage docfile interfaces and the object property set interfaces. In other OLE 2.0 systems (Chicago and Mac) the properties would only be available for objects stored in OLE 2.0 docfiles and accessed using the top level storage level docfile interfaces (same as Cairo). Some of our designs allow end user additions to this set of properties. Different designs have different schema mechanisms for properties. It may be possible to change the value of a property of an object in the passive state that will now be used when the object is reactivated. There are not strong guarantees that the passive state will validate changes the same way the active state does. Some designs (LMS and EMS) would not allow object activation for property changes because they don't support docfiles (also OLE 2.0 does not support replaceable docfile implementations). These approaches are very simple database-like and they don't do much to support the active object notions of encapsulation because they assume that the properties are available at all times.

It is interesting to note that there is a class of applications (Ren Views) that assume that all the object state can be expressed as (modifiable) properties. This is clearly naive for documents which can be very hierarchical structured objects, i.e., there might be a few properties, but the majority of the state must remain a BLOB. However, there are a class of objects for which very useful property view applications can be created. The Ren demos have many good examples of this. These applications are very similar to property page views. If we looked at how the design problems could be solved with OLE binding and activation, then we could probably figure out how to migrate and support this style of object customization application with the VB programming model.

CDE is trying to do the same things as Ren except they can do a little more because of the OLE characteristics of OFS and Cairo. Perhaps, for OFS object customization we could look at Ren as being the Phase 1 solution and build on that to get to the Phase 2 solution that is synchronized with Cairo. This would mean adding docfile support to LMS and EMS and a few other OLE support changes. This might be a better strategy than the current dueling banjos.

I think that there is a little too much design anarchy still remaining between all of our clients and system providers. We should have the same function or a comprehensible migration story. If we could sit down and sort through this, then I think we could have a reasonable strategy. The situation has been in design deadlock because the parties have been unwilling to compromise their requirements or ongoing implementation work.

I am assuming that the interfaces are random (arbitrarily different). I certainly can't figure it out from day to day due to the design instability of the various projects. Somebody give us good news and tell me that I am wrong. I want to be wrong about this.

(Active?) Document Library (DocLib) - I have not seen any of the design work for this, but it must be something that will track the versions of an object and support some notion of object comparison that can yield fine grained differences. It could be a passive design, but that might be even more difficult. This is a very special purpose interface.

(both) Explorer (Cairo) - The Explorer has a set of interfaces (a Cairo type) that objects should support if they want to act like Cairo "seams" in the Explorer. These interfaces are somewhat special purpose and require that the views fit into the Cairo Explorer pane architecture. There is no view and data separation in the design and the interfaces can not really be used effectively at the data level. This means that they are not useful for programming against by other clients.

(both) Queries (everybody) - This is where we finally start getting to BillG's favorite query scenarios. The query interface designs that have resulted from the different client requirements and data store implementations are all over the map. DAO 3.0 is trying to address some of this problem, but it requires a flattened passive data view on objects. This does not map well onto the internal structures of a document. There are other problems related the construction of queries. Certainly, it will be difficult for the average end user to construct these. (I could continue, but I am getting tired of writing. You are probably getting tired of reading.)

(active) Structured Objects (nobody?) - I pointed out some problems with the above two areas of client requirements. There is an alternative way to address both sets of problems. My description (of the requirements) will be somewhat rough, but I am sure that the details and problems can be resolved if we decide this is worth pursuing. (I am not going to design it or implement it. I might review it.)

1. Consider that container objects might have a set of "structured" objects that they can return. (We can find a better name for "structured" later). It should be possible to add to this set using the object customization features or the dynamic binding in the programmability model.

2. These "structured" objects have human readable names so that what they do is understandable (perhaps as the result sets from queries). There is an internationalization issue with this design.

3. Client tools can enumerate this set and find particular structured objects that they want to return. End users could choose from a list of possibilities if desired. Perhaps, as part of the interface, there is a way to specify an information type to be returned for the nodes of the structured objects. As an example, by using the IDispatch dynamic binding model it would even be possible to pass a fragment of code (moniker-like) that would evaluate to the name, view, or some other consistent object type.

4. It should be possible to create a small set of interesting viewers for the various structures that the objects could return. This gives us the view and data separation. The set of structures that "container" objects should be standardized, i.e., lists, tables, outlines, graphs, etc. We should keep the list short because we don't want to overwhelm ourselves with the possibilities.

5. It should be possible to make these structures composible. This would satisfy some of the hierarchical examples from BillG's email, categorization, and Navigator.

6. It is also possible to use the metamodel relationships to construct some of the structured objects that can be returned. This is certainly requires more infrastructural support that will be present in the future. This should not be an immediate goal for our objects.

7. Extending the above to be reasonable for passive objects should be simple in comparison to the above.

If this is not clear enough, I can verbally walk people through the above. I don't think that this is a particularly novel approach since we have solved some similar problems in a similar fashion, i.e., Explorer or Navigator. The difference is a couple of added degrees of freedom and layering in the design. The total combination should prove easier to live with.

The following is another relevant comment from one of my email messages that I wanted to repeat without spending the time to fold it into this memo.

> There are problems with taking a simple hierarchical view because many of the hierarchies that you might want to expose in the process of resolving a query may be virtual or computed. Changing an object can have many complex interactions with the various hierarchies and caches. Also, the problem can't be completely simplified to one of mapping storage into OFS. Which hierarchical relationships do you store? When the rubber meets the road we will have to have those types of semantics defined. Also, OFS has certain restrictions on its nodes with respect to naming. We can't expect all objects to have "names" that are meaningful to the end-user.

Metaphor Capsules - This is a very interesting product to look at closely for two very non-obvious reasons. As a product, it has somewhat limited value and appeal which means that it is not a strong revenue opportunity worth pursuing. First, it is important to understand the requirements it places on the architecture of the objects that the capsule operates on. These requirements are not unique to Metaphor. Second, it is important to think about how

Metaphor could be a more attractive product and what additional requirements that would generate on our document and data architectures.

Summary of the above - It is interesting to note the pattern that each client has determined that they need special purpose interfaces to perform their function. This is not inherently bad, since this is just what they all need. It is difficult to fault any specifics of the design work. Perhaps, one criticism could be related to object robustness and the active / passive inconsistencies that we have. These certainly project themselves out into the user model. The best of the above - queries and Metaphor - could use the object's programmability models as a last resort.

Look at this situation from our applications' or ISVs' point of view. They are being bombarded by special requests from everywhere. It is hard to prioritize the importance of anything and the priorities would be different depending on the platform! With this picture it is easy to see why OLE 2.0 could get the bandwidth of applications over the above. OLE 2.0 had a very consistent message with demonstrable value on all platforms. Maybe we need to step back and look at the big picture and perhaps develop a message that has broader understandability and consistent value.

How much of the design work should be in OLE 3.0/4.0?

This memo includes a number of items that could lead to new interfaces for inclusion as part of our systems, i.e., generic enough to be public. Other parts are more clearly part of Microsoft's product architectures. I won't answer the question here. It is an issue that needs ongoing attention over time.

Future Directions or Parallel Activities

I think that the component software technology direction will intensify in the future as more people learn how to write this type of software. We have seen some evidence of this from Claris Works, Software Publishing and Metaphor, OpenDoc, Digitalk Parts, and Borland. We should have some excess capacity to explore the technology further. I wrote about some of the general issues in my "APPA Mission and Notes" document. The architectural efforts could be made a little more concrete by one or both of the following activities.

New ground up component code base development - This effort should try to break some new architectural ground while maintaining compatibility with the Integrated Office architecture.

Single level store version of our components - The goal should be to understand how to move our implementations to a system that should offer the ultimate in object performance by using an object-oriented database engine for the persistence. The metamodel relationships present in our document architectures would allow an OODB to cache certain information that could be used by some of the queries (ones backed by real navigation).

Summary - What is a document?

This is an instance of a document. It satisfies my criterion that it be a container of organized information.

This memo is only a proposal for satisfying some high level objectives or requirements for Microsoft Integrated Office documents. It is neither a specification nor a development plan nor a strategy including a statement of synergy. The team implementing Integrated Office and the teams that want to work with Integrated Office components will have to produce those.