Inter-Office Memorandum

To	Wendell Schultz, Bob Metcalfe	Date	August 12, 1977
From	Peter Bishop	Location	Palo Alto
Subject	Comments on the Applications Software Functional Specification - Using character strings to identify files.	Organization	SDD

XEROX

Filed on: <Bishop>filenames.ears

I was surprised to note that the first release of the Janus Applications Software Functional Specification contains the concept of the "name" of a file that can be typed by the user in order to specify that an action be performed on the file. This concept leads to the requirements that the names of all the documents on the desktop be different and also that the names of all the documents in a file drawer be different. All of these effects seem to be quite innocent, but in actuality they have far reaching consequences that will hurt the system we are developing now and will also impose an inferior structure on future systems.

Experience with computer system design has shown that character string names should not be used by programs to name files. As a result, Pilot will not support character string names for files. Pilot will support naming a file by a FileID: a 64 bit name that is unique for all time and is not tied to a physical location [see the Pilot Concepts and Facilities, page 60]. Thus whenever a program manipulates a file, it must have the file's FileID. This is achieved by storing a FileID as well as a character string name whenever there is a reference to a file. If the user can type the name of a file in order to refer to it, however, then a program must obtain a file without having its FileID. If, on the other hand, the user can only refer to a file by pointing to the file or by pointing to another reference to the file, then the program will be able to obtain a FileID for the file. It turns out to be easy to design a system that is convenient to use but in which the user never types a file name in order to specify a file. The advantage of this way of operating is that it allows the user to benefit from the advantages of using FileIDs to refer to files without having any restrictions placed on the character strings that the system displays to describe a file to the user.

Character String Names Do Not Work Well

The ability to point to character strings or icons displayed on the screen by the system has not been available on many computer systems, so users have been forced to specify objects by typing character strings that name them. Communicating with a computer in this way works reasonably well in programming languages, but begins to break down when used to specify files.

One of the problems arises because of the way people use words and names in real life. In real life, a conversation always takes place in a context. A precise specification of what the context is, however, is difficult to provide because the exact context is very dynamic. Thus the sentence: "The bus just turned around," can have different meanings depending upon whether the conversation is occurring on the street, whether one of the participants in the conversation just talked about a bus, or whether the participants in the conversation have been having a prolonged discussion (over several days) about a bus. Many factors can affect the context in which a discussion occurs. It is not clear that anyone really understands all of the ways in which the context of human communication can be affected. One problem that is inherent in such dynamic contexts is that the different participants in a discussion may have different ideas about what the context is. To handle this, people are very good at handling ambiguity.

Whenever we construct naming structures involving character strings we must realize that the user is at home with such complexity and is used to dealing with other people who also are able to handle such complexity. Unfortunately, current computers are not able to handle such complexity, especially since it is not fully understood. The theory behind the use of character string names for files is that a directory (i.e. the desktop or a file drawer) is a context in which a certain set of names are defined. Unfortunately, it is usually difficult for the user to remember what the context of the directory is. Often a directory contains names that the user thinks of as belonging to several different contexts. As a result, the user must invent names that are not very context dependent. This becomes even more true if a file keeps the same name when it is moved between the desktop and the file drawer and if the name must be unique in both places. These problems are solved reasonably well if files are given long names that are intended to describe the contents of the file. It is not convenient, however, for the user to type names that are long enough to adequately describe the file. Furthermore, no character string can be guaranteed to be unique over very wide contexts. This does not bother people, because people are used to dealing with such naming conflicts, but if the system is identifying a file by its character string name, it can get confused and do things that the user does not expect. An advantage of a FileID is that it can be guaranteed to be unique over very wide contexts and so can be used in places where character string names cannot be used at all.

Thus the specific problems with using character string names are:

1) There is an inherent conflict over whether the name should be long or short. Long names allow the users to tell what files are in a directory just by looking at the names, while short names make it convenient for the user to type the name. In practice a compromise is made that satisfies neither criterion.

2) Users would like to use the same name in different contexts even if there is a naming conflict because the user is accustomed to thinking of the file with that name and is able to use adjectives to avoid the naming conflict.

3) Requiring different names on all the files in a directory makes it awkward to deal with multiple versions of the same document. Especially with the display of file cabinets that has been proposed, the user could use the date of creation or the size of the file to distinguish between the different versions of a document. There are instances where it is useful to have different names for different versions of a document, but this is a policy decision that should not be forced by extraneous considerations.

4) A directory does not correspond to a context that is meaningful to the user.

FileIDs Provide a Superior Name Space

FileIDs are a better way for a program to name files. This occurs for several reasons:

1) FileIDs are unique for all time, and so the same name can be used for a file regardless of what context the program is in.

2) FileIDs, being a constant, small size, are more convenient for programs to manipulate than character strings.

3) FileIDs handle multiple versions very easily: each version has a different FileID.

4) Programs need never have added complexity to handle the case of what to do if there is a name conflict with the file.

5) Programs sometimes want to be able to bind themselves to a particular version of a file. If the file system only supports character string names, this is impossible to specify, while if FileIDs are used, especially in conjunction with the concept of immutable files, then the FileID specifies a specific version.

Problems of Using FileIDs

Perhaps the most serious problem associated with using FileIDs is that the user should never see a FileID printed or ever need to type one. This can be achieved, while still causing the user to specify a FileID whenever the user specifies a file. If each piece of text that the system displays and that describes a file has the file's FileID associated with it, then when the user points to the text, the associated FileID is available. When a user types text that refers to a file (the text will exist in a human context that need not be understood by the system), the user must also specify what file is actually intended either by pointing to the file itself or by pointing to another text string that refers to the file and has a FileID associated with it. It will be easy for users to understand the concept that a string of text refers to a file not because of a computation based on the character string, but because eventually a person pointed to the file that was really meant. The user will come to understand that some character strings that talk about a file cannot be used to obtain the file because the person who typed the string did not point to the file, while if the person who typed the string did go to the extra effort of pointing to the file, the user can see the file just by pointing to the character string and depressing the DISPLAY DOCUMENT key. Thus when the user receives a letter over the system that says "... as per your letter of the 23rd ...", the user may then be able to see the appropriate letter easily without having to have named the file "your letter of the 23rd", or indeed to even remember *what* name the user gave the file containing that letter.

Thus although there seem to be problems with using FileIDs, a style of operating can be developed in which FileIDs are in fact used by the system to name files, but no restrictions are placed on the techniques people use to describe files or on the contexts that people can dynamically create among themselves for describing files.

Other Advantages of Relying on FileIDs

Initially, we expect the user to have to type the descriptive text for a file, but once we have eliminated the requirement that names be unique in certain contexts, it will be possible for software to generate the descriptive text automatically from the contents of the file. Thus we will not only have eliminated the typing of names to identify files, but we may be able to eliminate the typing of the descriptive text used by file drawers.

Once Pilot has provided FileIDs, we should make an attempt to use these FileIDs directly without requiring another level of name lookup. We will gain in efficiency because we will not need extra lookup software in core nor will we need the extra data bases that perform the extra level of naming. This software and its data bases will still exist, but they will be removed from the mainstream of computation. In addition, they can be tailored to support information retrieval better, since this will be their major use.

In order to really take advantage of FileIDs, however, we must ensure that users specify FileIDs without typing them into the system. This requires a different style of operating than that used when names can be typed directly. If we introduce the concept that users can type file names directly in order to refer to a file, then we will never be able to change to a style of operating that can make full use of FileIDs. If, on the other hand, we do not allow users to type file names now, and it later turns out to be a good idea, it will be easy to add this extra operation at that time. Presumably we would then understand how to allow the user to type file names without losing the advantages provided by FileIDs. It would be tragic if we built a system based on FileIDs that did not take advantage of their possibilities.

Specific Problems in the Functional Specifications

There are a few places in the Functional Specifications that allow users to type the name of a file. I will show how these functions can be performed without typing the name of a file.

Abbreviation Objects

When editing a document, a user has a number of abbreviations available to him that can be invoked by typing the name of the abbreviation and then the EXPAND key. The definition of the abbreviation is looked up and the abbreviation name is replaced by the definition. This is a very valuable function that allows users to achieve faster typing speeds than can be achieved with standard typewriters. There are some deficiencies in the mechanism as proposed, however. First, there is no way for the user to find out what abbreviations he has defined, second, the user is unaware of how much storage is being used to store the abbreviation definitions, and third, there is no way for the user to work in different abbreviation contexts. All of these deficiencies can be corrected by creating the concept of an *abbreviation object.* This is an object that contains abbreviation definitions. Abbreviation objects can exist on the desktop or in file drawers. Looking inside an abbreviation object allows the user to find all the abbreviation definitions and also to edit the abbreviation names and their expansions. The abbreviations contained in an abbreviation object are not available for use unless the abbreviation object is on the desktop and is *activated.*

An extension of abbreviation expansion described in the Functional Specifications is the *include segment* operation. If the user types an abbreviation followed by the EXPAND key, but the abbreviation has not been defined, then the system searches first the desktop and then the file drawer for a file whose name is the same as the abbreviation whose definition is being searched for. If such a file is found, the entire file is included in the document. This feature has several unfortunate effects. First, it causes the user to have the idea that files can be specified by typing their names, and second, whenever the user mistypes an abbreviation name, a good deal of disk activity will be caused in searching the various directories that are mounted. If, by some chance, a file of that name is found, it will be copied into the document. If we accept the argument that there are times when it is desirable to include an entire document in another document (rather than referring from one document to another), these problems could be solved by making use of our abbreviation object. We should allow the user to define an abbreviation that specifies an entire document. When the user creates the definition, the user must point to the appropriate document or to a character string that has the file's FileID associated with it (such as the file's entry in the file drawer). When the user finally wants to include the file in a document, he will specify a name that is reasonably short, as are the other abbreviation names, rather than being forced to type the long string of descriptive text that usually is displayed in the file drawer.

Selecting Files in the File Drawer

Another place where the user is encouraged to type the name of a file is when browsing in the file drawer. The user may select a file by pointing to a file name and depressing the FINE mouse button. Once a file has been selected, various operations can be performed on it. A user can also select a file by typing its name, however. Curiously, however, a user does not seem to be able to search a file drawer for a particular character string. If this ability were added and the ability to select a file by typing its entire name removed, then it would not be necessary to prohibit two files from having the same name and the user would not be tempted to think that the system understands file names. The user would depress the SEARCH key, type the string to be found, and the file drawer would be searched from the current selection. If the text appeared as part of a name or a date, then the first file found with that text embedded in it would be selected and the display would be scrolled to show the file that had been selected. This would be much more powerful than requiring the user to type the full name of the file.

Consideration of this problem leads naturally to the suspicion that the technique of displaying the name of the selected file in the current file selection field may by itself suggest to the user that a file can be specified directly from its name. I am not sure this follows, however, and I am not aware of the rationale behind the document selection field in the file drawer. Although it seems nice to be able to tell what document is selected even if it has been scrolled off the screen, wouldn't it be better to be able to select more than one document at once and then have a mechanism that displays all the selected documents?

Conclusion

We are constructing a system that has a pointing device and whose operating system supports the naming of files via a very powerful name space consisting of unique IDs. Such a system provides us with many powerful possibilities. To realize these possibilities fully, we must prevent the user from having the concept that the system understands what the user means when the user types the name of a file. When the places in the Functional Specification that encourage the user to have this concept are investigated, it is found that these parts of the design are deficient in other respects as well. When the ability for the user to type the name of a file in order to specify the file is removed, improvements are made in the design of the system, often allowing the amount of typing required of the user to be decreased. Although we may, at some time in the future, want the user to be able to specify a file by typing its name, there is no need to introduce this concept now. Indeed, if this concept is introduced prematurely, it will prevent us from taking advantage of the FileIDs supported by Pilot and will require another layer of naming software and data structure to exist in core. We want the best of both worlds, not the worst of both worlds.