DEFINING A SYSTEM INTERFACE

What do we want from a standard interface ? Several things, but the unifying theme is the need for a picture of the operating system (the system mental model) that's simple and understandable. We decided that one important factor in achieving this end is *consistency.* We also want our system to be comprehensible and helpful, but in some ways consistency is the most obvious place to start because it's comparatively easy to check. Though most commonly thought of as attributes of user interfaces, in fact we'd like these principles to apply everywhere in the system, because all the parts of the system are used by someone – ordinary user, systems programmer, administrator, or whoever – and the same criteria apply.

What's implied by this requirement of consistency ? The important characteristic is sometimes called the *principle of least surprise* : as far as possible, we'd like things to work just as we expect them to by analogy with other things we know about. For example, we want filenames always to have the same form, and to be interpreted in the same way, we want special key (or mouse, etc.) sequences always to do the same thing, and so on. Generally, *we want to avoid special cases.*

Translating this excellent principle into practice in every part of the system turns out to be very hard indeed. We do not propose to analyse all the possible complications, but will illustrate the problems with some examples.

CONSISTENCY IN DEVELOPMENT : THE APPLICATION PROGRAMMER INTERFACE.

Within the operating system, that's largely a matter of design discipline. The same action must always be done in the same way – which means that we must provide a single procedure to do it, and insist that it always be used. We could, for example, have a single procedure to parse file names, which will guarantee consistent treatment throughout the system. (Notice how this leads to the idea of servers of various sorts, even down to levels of tiny detail in the system. It's also a good area for object-oriented implementations.)

Outside the operating system, that won't work. We can't force independent developers to write their software according to some arbitrary conventions we've dreamed up – the best we can do is to persuade them that it's in their interests to follow our lead. We do that by making our interface so attractive that people want to use it, and by making available to the developers the standard procedures that we use to write the operating system. The result is commonly called an *application programme interface* – and more commonly called an API. The traditional proprietary operating systems never succeeded particularly well in providing useful APIs (they might not have tried very hard), but both Unix and the Macintosh system have done much better.

How do you go about deciding what standard procedures you need in your API ? You have to analyse the tasks to be done, and determine what depends on what, and which bits would be useful. Then you have to do a lot of thinking.

Suppose, for example, that your system (like most systems) uses a lot of filenames presented as character sequences. (We use a textual example for the moment only because it's rather easier to present in written form; some parallel comments on graphical interfaces appear later.) It would clearly be a good idea to write a standard filename parser which could be used to guarantee consistent treatment within the system, and to make it available in the API. Just what you want in such a parser depends on how the filenames are used, and what's expected in different contexts. Consider a few cases :

print <filename> : We have to look for the file, and find out whether it's there. If it is, and it's printable (not a directory nor a code file, for example), then it must be printed. If not, we need to return a plausible error message.

print <filen*> : We have to look for a file with a name which begins with "filen", or several such files, and return a list of zero or more results.

edit <filename> : Rather like **print**, but if the file isn't present we can either regard it as an error or we can assume that we have to make a new file.

open <filename> for reading : We must search for the file, and report an error if it isn't there.

open <filename> for writing : We must search for the file, and report an error if it *is* there – or overwrite the existing file, or append to it, but anyway we need to know.

To do a proper job, we should study all the places where filenames are used, and analyse them much more carefully than we have done here, but it is clear that there are several different ways of using filenames. It also seems that searching for a file name is usually associated with checking for its existence, so perhaps the two operations should be combined.

The examples are reassuring in that it doesn't seem that there are going to be any big surprises in the parsing itself, so we can proceed to design it. It would probably be sensible to provide a filename parser which could accept a string and identify a filename at the head of the string. What should it return ? Here are some suggestions :

- something to say how far along the string the filename goes : that seems fairly uncontroversial, as you have to know where the filename ends so that you can continue to parse the rest of the text, if there is any. Should the procedure return a pointer to the end of the filename ? or the original string with the filename removed ? or is the length of the filename sufficient ? or should it be given a pointer which it automatically advances past the filename ? An interesting device used with functional languages but possible in principle with at least some others is for every such procedure to return a *continuation* – something amounting to another procedure, which you use if you want to continue. This is quite a versatile trick – for example, you could also use it when expanding wildcard characters.

- an indication of whether or not it found a filename : again, an obvious requirement. But what if the parser doesn't find a filename ? – should it produce an error message ? If it does, then the message will always be the same, which is in line with our aim of consistency – but the absence of a filename might not be an error. (Perhaps just "edit" without a filename is an instruction to start a new file, or an absent filename implies that you should use the terminal or screen, as is often assumed in some contexts in Unix systems.) Further, an error message composed by such a low level procedure is unlikely to be helpful. It could say "No file name found" – but if we defer the message production until execution gets back to a higher level, we could perhaps say "Couldn't find the name of the file to be copied", which would be much more helpful. Perhaps it would be better to leave the error message to some other procedure – but that's making the system less simple.

- the filename itself in some form : once more, sensible enough. But how should it return the filename ? As a string, or a pointer to the original string ? That means you have to parse it again to find the structure. As some sort of structure, then ? What sort of structure ? And suppose you want to provide "wildcard" filename expansion ? Should that be expanded within the parsing procedure ? If not, the programme will have to deal with it every time a file name is used. But you can't expand the name if you can't see the directory, so you couldn't use the procedure with filenames written now but intended to be used in some other context. If you do expand the wild characters, how are you going to return the list of names ?

- an indication of whether or not it could find the file : not quite so obvious. The examples suggest that we often want to know, so it would simplify the system to combine the parsing and the checking, but if it isn't always possible – as with a file name used out context – then that's that, but again the system becomes more complicated.

Things are evidently not as simple as we might have hoped – but perhaps that's because of the rather cumbersome reliance on text input. What about systems which rely on graphics interfaces ? Well, to begin with, you still have to address many of the same problems. Files are still likely to be identified by real names somewhere in the software, so many of the problems are (it seems) inescapable. ("It seems" is just to play safe; so far we know of no significant system which has escaped ordinary file names altogether, but perhaps there's a way no one has thought of yet.)

But there are also plenty of problems to do with managing the different style of interface. Something has to convert the signals from the mouse into a position on the screen, and move a pointer on the screen accordingly. Something has to know enough about the screen layout to be able to work out what process currently owns the point on the screen on which the pointer is resting, and – if it's a file icon or file name in a table – to associate the position with the correct file. These are the screen equivalents of parsing file names.

Then something has to know what to do with a click (or a double-click, or whatever). Should the operating system do anything, or just pass on the event to some other process ? If the operating system has to take action, what is it ? Is it purely internal, or does it require some change to the display ? These are the screen equivalents of interpreting the instructions.

There is no escape. The process is governed by no laws of nature, so everything is artificial, and must therefore be designed. Typically, there is no right answer (though there might well be a lot of wrong answers), so you have to select one set of conventions and operations which you believe will do the job. Whichever way you do it, it is difficult to design a satisfactory set of procedures for universal use. You can try to minimise the number of procedures in the hope of simplifying the API, but then to ensure that you get a flexible system each procedure will have to cover a lot of actions, and will have to be provided with many parameters (or something equivalent) to select the right one. Alternatively, you can try to minimise the numbers of parameters to procedures, again in the hope of simplifying the API, but then you'll probably have to provide many more procedures. Comprehensibility suffers either way. There are lots of decisions to make, and the whole package has to be reasonably easy to use if people aren't to give it up as not worth the effort.

And that in turn means *documentation*. If people don't like your system, they may choose not to use it; but if you don't tell them how to, they *can't* use it. Complete, effective, and comprehensible documentation is essential if you want your interface to be accepted.

CONSISTENCY IN USE : THE USER INTERFACE.

The user interface has evolved over the years from nothing at all, through a necessary evil hardly worthy of serious design, into the most intensively designed part of the system. That's not to say that it's necessarily the best designed part of the system – just that it's still not very well understood, and a powerful selling point.

Consistency in the user interface is obviously a good idea, but by no means easy to achieve. Here are some examples to illustrate the minefield you get into if you aim at a consistent interface. We begin with a textual interface.

First problem : what does consistency mean in the context ?

First answer : a particular sequence of operations should always have exactly the same effect.

First comment : impossible. In Unix, **rm x** totally and irretrievably destroys a file called **x**. Your terminal "is" a file called **/dev/tty** : what should **rm /dev/tty** do ? We can change the definition so that **rm** destroys only a destroyable file (which also takes account of protection, and some other complications), but after you've

found a few exceptions and patched the definitions to fit you end up with something so complicated as to be not much use anyway. So let's redefine consistency.

Second answer : a particular sequence of operations should always do much the same sort of thing.

Second comment : that's better. It gives you a useful guideline for deciding whether or not the operation does what you want, and if there's any doubt you can always look up the details. (If you can find the manual; and if you can understand it – but that's another problem.) It also leaves room for explanatory messages in case you ask for something impossible. Unix renames files if so directed with the instruction **mv <name1> <name2>**. **mv** might seem to be an odd abbreviation for "rename"; it's actually an abbreviation for "move", which is mildly appropriate because by using extended names for **<name1>** and **<name2>** the result appears to be to move a file from one directory to another. It's still misleading, as it's all done by shuffling directory entries, and the file doesn't move at all. Because of that, when a move really *is* required – which is so if **<name1>** and **<name2>** must be stored on different discs – **mv** won't work. That hardly conforms to the principle of least surprise, but at least it can give you an error message, which we can now fit into our definition of consistency.

Third comment : there is more to **mv**. Suppose you give the instruction **mv A B**; what should happen ? Obviously, if there's a file called **A** it should be renamed **B**. But suppose there's already a file called **B** : what then is "something quite like" the basic renaming operation ? Should the system replace the existing **B**, or should it report an error ? That depends on the judgment of the operating system's designer. (If there is one.) Suppose instead that there's an existing directory called **B** : again we can ask whether the system should replace it (rather drastic !) or report it. In fact, Unix does neither of these; instead, it moves the file **A** into the directory **B**, as if you had instructed **mv A B/A**. Is that consistent behaviour ?

What about a GUI system ? You might expect misbehaviour from Unix, which was designed before people understood the importance of a consistent interface – but surely the Macintosh changed all that ? Ha ! The Macintosh has trouble with its equivalent of **mv** too. What happens when you drag an icon from one folder to another ? If the two folders live on the same disc, the icon moves; if they live on different discs, the file is copied. That certainly gave one of us trouble when he moved up from a smallish system with floppy disc and RAM disc to a bigger system with a hard disc sufficiently large that he didn't need a RAM disc any more. The result was that a lot of operations which had previously been nice safe copies between floppy disc and RAM disc turned into moves, and he didn't always notice until he came to look for the files again. That is *not* consistent behaviour.

ADDITIONAL, RATHER UNFAIR, BUT NOT ENTIRELY POINTLESS COMMENT.

The trouble with some, at least, of these decisions is that there isn't an obviously right answer. Macintosh and Unix come down on opposite sides of the fence in deciding how to move a file from one disc to another. That's an arbitrary decision. But it's hard to see how the Macintosh system could in any way combine its move with a rename instruction, as Unix does : that's a forced decision, dictated by the nature of the interface itself.

Well, perhaps it's a little hard expecting similar performance from two such dissimilar systems, but with the current interest in standards which extend over all systems it illustrates a difficulty. Even with apparently similar interfaces, could there be characteristics of the underlying systems which made it difficult or impossible to transfer the same actions without changing the meaning ? We don't know an answer to that question, but it is one reason to emphasise the separation between the interface software and the system underneath.

QUESTIONS.

Consider some operations on window interfaces. Do they have the same meanings in all contexts ?

Why do so few textual systems allow you to correct and resubmit an instruction ? (Unix provides a way of editing and resubmitting, which works if you can remember how to do it.)

Is it possible, in principle or in practice, to correct and resubmit an instruction given with a GUI system ?
