SETTING CONVENTIONS

One of the characteristics of an operating system is that, in order to ensure that it can communicate with people who want to use it, certain conventions have to be established. This happens at all levels of the system at which links with external entities might be required – so software developers will be required to follow certain rules if they are to write software which will be compatible with the system (now often expressed as defined APIs), and operations staff will have to conform to certain procedures when managing the system.

In just the same way, people who use the computer system to get their work done must follow certain rules, some of which are more obvious than others. They must learn what actions to perform to convey their intentions to the computer system, and they must learn how to interpret what they observe of the computer system's behaviour. In effect, they have to learn two languages (though the "words" might be composed of points, clicks, and drags or of icons' colours and menus' visibilities instead of letters).

It is from their experiences in learning and using these "languages" that most people build up their beliefs about the computer system, so it's important to design the languages in such a way that people end up with useful beliefs.

THE *SYSTEM METAPHOR* (or *SYSTEM ILLUSION*) AND *SYSTEM MODEL* (or *SYSTEM GENIE* or *SYSTEM DOCTRINE* or *SYSTEM IMAGE*) .

(Those are all (non-standard) names for (respectively) the image which the system is designed to present, and the collection of ideas you acquire about the system as you learn to use it effectively. No, we didn't invent them; we've read them all in technical literature.)

The system metaphor is a matter of design. It is more prominent in recently developed systems, because the idea that a computer might be likened to anything but a computer is of fairly recent origin. By far the best known metaphor is the *desktop metaphor*, familiar through the Macintosh and Windows systems though originally developed by others (see the previous chapter).

Everyone who uses a computer system for any length of time builds up a mental model of what the system "is", and a way of imagining its mechanism. (We shall call it the system model, though for a long time we preferred to call it the Genie, because that term is defined[REQ7] to include one's perception of the system manual, the "help" facilities, and do on. We have changed our terminology because we were so spectacularly out of step that no one understood what we meant, but we continue to include those matters within the system model.). The model might be wildly unrealistic – but that doesn't matter too much so long as it works. We prescribe the stage 2 courses as prerequisites for 340 not so much for any specific topics in the stage 2 material, but as an approximation to a guarantee that you'll have a reasonably realistic system model.

The metaphor is under the designer's control; the model is not. It is the aim of good interface design to be conducive to the development of an effective model, which represents the system well enough to lead its owner on to correct inferences about how to use parts of the system which are new. A good model is a good analogue of the behaviour of the system, so that expectations based on the model run parallel to the actual behaviour of the system. Good system design leads to a model which is easy to learn and to use. A large part of this design concerns the control language – that goes a long way to determining how people see the system.

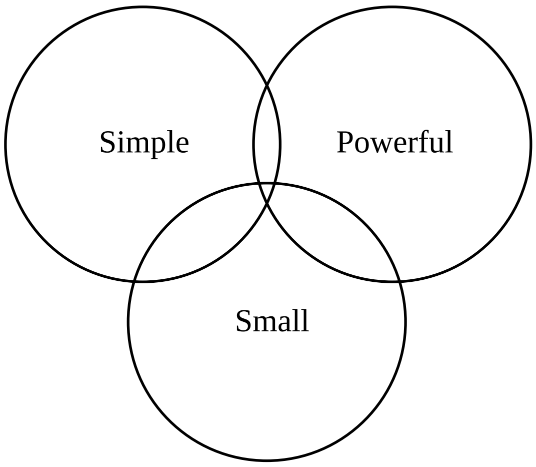
DESIRABLE FEATURES.

The aim is always to help people using the computer system to get on with their work. Accepting that a completely invisible interface is probably impossible to construct, we would like the interface to be (on the negative side) as little of a hindrance as possible, and (on the positive side) helpful wherever it can be.

We can reduce the hindrance by looking for the difficulties which people experience when using a computer, and seeking ways to alleviate them. For example, we might try to remove technical vocabulary from the instructions, or avoid the requirement for typing skills by seeking a replacement for the keyboard.

We can be helpful by providing instruction and information which is easily accessible when it's required, and by making the system simple in structure so that once people learn something about it they will be able to work out how to carry out other tasks.

Generally, we should try to build a system which is sufficiently powerful to do the job required, but, given that, is as small and simple as possible, so making it easy to learn, use, and understand. This (free impression of a) Venn diagram depicting the relationships between the sets of small, simple, and powerful systems is the next stage in our discussion :



That's supposed to illustrate the unfortunate fact that, by and large, the three desirable attributes are mutually antagonistic. A powerful system is not naturally simple or small; a small machine might be simple, but if so it is unlikely to be powerful; a simple and powerful machine is unlikely to be small; and so on. Our desirable systems are those which lie in the small triangle in the intersection of all three sets. We can conclude that it won't be easy to construct operating systems which conform to our desirable pattern.

What we can do, though, is set ourselves system design goals which should lead us towards the target. We want a system to do as much as possible with as little effort as possible on our part : we'd like consistency, comprehensibility, helpfulness.

CONSISTENCY.

An interface is said to be consistent if the consequence of each possible input action is similar no matter what the system is doing at the time. The consequences need not be identical in different circumstances, but they should be recognisably and plausibly related. For example, the same action (like delete something, choose something, move something) should be done in the same way everywhere it's done. Conversely, the same action should produce the same result everywhere it's done, which is not quite the same thing. It is generally good to stick to any established pattern of behaviour or form of instruction, avoiding special cases. If we can incorporate this property into our system, people will have fewer new ideas to learn, and each idea will cover more cases. In practice, the restriction to "the *same* action" and "the *same* result" turns out to be too constraining, as different things are rarely the same; though there's an obvious connection between the ideas, deleting a piece of text from a string is not the same as deleting a file from a disc, and some differences are to be expected. Nevertheless, the more things we can group together into common categories, the less we'll have to learn and to remember, so it's better to extend the principle to "similar things should be done in similar ways".

Consistency is important because it simplifies the interface. Someone who uses a consistent interface has to remember how to perform a few classes of actions which can be used in most circumstances, each class covering a set of loosely related operations which are effected in correspondingly related ways; with an inconsistent interface, each individual operation could require an action which bore only accidental relationships with any other, so the burden of facts to be remembered could be much heavier. In a "modeless" system (which could better be called a "single-mode" system), there is only one set of responses no matter what programme is being executed. The system is therefore consistent.

To make this happen isn't easy, as it requires stringent internal discipline while constructing the system. Guidelines, standards, and conventions must be established and rigidly observed; you have to produce a rulebook for your development team (even if it's a team of one – it is all too easy to revise your ideas of what you really intended to do in the light of experience, without realising that you've shifted ground.), and you have to provide a police force to ensure that the rules really are followed. So far as it's possible, the desirable guidelines etc. should be implemented as programme modules of whatever sort is currently fashionable (say, objects), and always used – but you soon find that designing a satisfactory set of modules is a very difficult job in itself.

Here's a comment[REQ8] about IBM's notions of consistency in interface design. It's from some years ago, and it isn't clear to us that practice has yet followed design to quite the extent implied in these excerpts. That's not intended as another nasty anti-IBM comment; it isn't unreasonable to see these good intentions as early moves in an operation which is still continuing, and not only in IBM. (We wish that someone would tell Microsoft about it. Their versions of their own Word software differ from each other in quite puzzling and confusing ways.) But here's the comment on IBM's version :

One thing is certain about Systems Application Architecture – it's important to IBM. The world's largest computer company is said to take the idea so seriously that any division wishing to develop a product that doesn't comply with SAA's rules has to be given special dispensation.

On one level, SAA sounds simple. IBM defines it as the framework for developing applications across its disparate hardware ranges. Applications written to SAA specifications should look and feel the same, whether they run on a PS/2 or a mainframe. In addition, an application written for a PS/2 should also run on an AS/400.

As it stands today, SAA is a confection of co-operative processing and common user access, liberally sprinkled with the idea of hardware independence. But cua remains the most appetising ingredient. Even those like Mike Moore, vice chairman of the IBM Computer Users' Association and something of an SAA sceptic, can see the usefulness of a common interface across programs and machines. 'It would be nice to have the same function using the same key', he says, explaining that, at present, the PS/2's help key is F1, the System 36's is PF6, the Wordperfect wordprocessor's is PF3, and the 4300's is PF8.

(No, not *that* Mike Moore.) And here's IBM's definition[REQ9] of their Systems Application Architecture :

IBM describes SAA as the framework for developing consistent, integrated applications across its OS/2 Extended Edition, AS/400, VM and MVS environments.

It is made up of four components:

- Common user access defines the look and feel of a screen and keyboard, whether attached to a PS/2, AS/400 or 370 mainframe .
- Common programming interface defines the languages, commands and calls that programmers can employ to create applications which can run under any of the SAA environments.
- Common communications support defines the consistent implementation of data streams, application and session services, network and data link controls.
- Common applications are applications that use common user access, the common programming interface and common communications support and therefore run under OS/2 Extended Edition, OS/400, VM or MVS.

THE SCOPE OF CONSISTENCY.

The ideas embodied in the quotations above illustrate large scale consistency, where the aim is to make it easier for people to move from system to system. This is a fairly recent idea; to some extent it has only become useful recently, as in earlier days very few people used more than one computer system at all regularly. There are at least two other sorts of consistency, distinguished by the extent over which the consistent behaviour is intended to be found.

There is a larger scale, in which the aim is consistency over the whole computer industry. Such consistency has not been a feature of computing in the past. If you were an IBMer, then you were an IBMer and – usually – you didn't know much about anything else. The IBM effect was special because of the sheer size of IBM and the number of its installations, but the same happened with other systems. It was particularly galling for people accustomed to other systems for the good features of their systems to be ignored in the trade (perhaps understandably) and in academic circles (where they should have been more careful, but it *is* difficult to keep up with all the developments). More recently, there has been a move towards the idea of *open systems*, mainly characterised by the publication of APIs (like Posix, which we met in the *HISTORY* section) as standards to which manufacturers are expected – or, at least, encouraged – to adhere.

Most commonly, though, we think of consistency on a comparatively small scale within a particular computer system. Most systems have more or less standard ways of doing things, if only because to make everything different is almost as hard as making everything the same, and the easy way is to do it in much the same way as you did it last time, but consistency wasn't really emphasised as a desirable thing until graphical interfaces came along. Perhaps the most influential system in this area has been the Apple Macintosh system; it wasn't the first system with a graphical interface, but it was the first to become significantly widely used, and the designers took the user interface very seriously and did it rather well. The rest of our examples here are taken from that, and similar systems.

COMPREHENSIBILITY.

The system should be designed so that it helps people to construct a simple mental model of what goes on in the system. (That's the system model we mentioned.) A very significant development in user interface design occurred when it was realised that the model doesn't have to be based on a computer; other models can be used.

TRADITIONAL – the system looks like a computer. Explanations are presented in terms of discs, files, memory … – so people need to learn about computers.

Disadvantage : more work, not actually very simple, people get put off.
Advantage : it's something new, so people aren't confused by preconceptions.

MORE RECENT – the system looks like something else – like a desktop (Macintosh), office (Word processors), database (Pick). Explanations are presented in terms of documents, folders, and other common objects, which behave in ways that people already understand.

Advantage : it's familiar, so people already know something about it, don't have to learn new things.
Disadvantage : it's a lot harder to make it work – and it's not true, so somewhere the metaphor will break down.

Whichever way you do it, it's hard. We've said something about the textual languages in *JOB-CONTROL LANGUAGES* earlier; the difficulty there is to express the actions to be performed and the objects to be used in a reasonably natural way, and then to write a language interpreter which can understand it. The designer of a graphical interface has more scope in many ways, but there is still a lot to learn when you first start using such a system. (If you've been using a graphical interface for a long time, you might find that hard to believe, but if you watch a real beginner for a while you'll see that it's true. The clever thing about the graphical system – assuming it's a good one – is that, because some of these interface design guidelines have been observed, it's comparatively easy to build up a workable system model fairly quickly.)

We mentioned the Pick system; here's a very brief description[REQ9] of it :

Pick is a multi-user, multi-platform operating system with its own built-in relational database. It can also be used as an applications generator.

Its database is particularly efficient because, unlike other rdbms, it accesses the hardware direct, rather than via the operating system.

Everything in Pick is held in data files, each of which has an associated dictionary. This defines the content of the file and its link to other files.

The organisation of files is pyramidical, with data files at the bottom and the master system at the top. Between the two is a master dictionary which defines what operating system commands are available to each user.

By editing the dictionaries, no data file need become redundant, nor need the same data be entered twice.

But Pick's main attraction is its ease of use. Because even its database query language is recognisably close to English commands, a relative novice can manipulate data with the a minimum of training.

For years Pick slugged it out with Unix for the position of premier operating system in the open systems market. It was another case of David taking on Goliath, only this time the giant won.

It's a textual system, but notice the remarks on the ease of use of its language.

HELPFULNESS.

Ideally, the system should be self-explanatory – if you don't know what to do, you should be able to find out easily from the system itself. That's one of the reasons for designing the system metaphor in such a way as to encourage the development of a good mental model : the models people make of the system should be sufficiently reliable to suggest the right answers.

INSTRUCTIONS should be available (can you look at a list of system instructions ?), clear (do the names mean anything ?), complete (can you do things that aren't evident from the instruction names – maybe as variants of other instructions ?).

That's comparatively easy to manage – which isn't to say that it's often done – with a textual system, in which there *are* visible instructions which can be listed and defined. It's not nearly as easy with a graphical interface, where instructions might be given by motions rather than words. When printing is initiated by dragging the icon of a file to be printed to the icon of the printer, where can you look it up ? If you don't know the answer, it can be quite hard to find. That's why a simple system model is essential to a graphical interface.

"When all else fails, read the manual" is a well known proverb of computing which is, unfortunately, supposed to be a joke. Particularly when you can't easily show how to do things on the screen, a complete manual with a good index is very valuable. It is true that one tries to construct the system so that recourse to a manual is rarely necessary – but that only means that the manual must really work when the rare occasion turns up, because whoever's using it won't be familiar with it.

HELP should be available from the system as material displayed on the terminal screen. We'll go into more detail on the help system later, but it's appropriate here to notice that, once again, it might be easier to give with a textual system than with a graphical one – or, perhaps more accurately, textual systems are commonly constructed to give better help.

That's rather odd, as one might expect that the potential of a graphical display to attract someone's attention to the occurrence of an error, and to present information about it, would surpass that of the humble textual interface. All too often, though, an error is signalled by the appearance of an uninformative message, after which the system restarts. In the unusual case of an informative error message, the helpful information might well vanish when you try to inspect the window of the programme which caused the error.

CAN IT BE DONE ?

The answer seems to be "Yes, to some degree, but with difficulty". It is certainly true that computers are far easier to use nowadays than they were in the early days of microcomputers, and that the microcomputers were usually much easier to use than the traditional mainframe systems. Interfaces are more helpful, in that they give more information; they are probably more consistent, in that they rely much more on a generally understood vocabulary of pointing and clicking to select items from displays of icons or from menus. It is not so obvious that they are more comprehensible. The reliance on icons rather than words is at a level which can be disconcerting even for an experienced computist when moving to a new system. Once you know the system, it's easy – but so were the old mainframe systems. We discuss two examples to show that not all is sweetness and light.

One of the major features of the new interfaces is the pointer, and its use in selection. There is general consistency in the convention that there shall be a pointer, and tat it shall be moved about by a mouse or something equivalent. (Other means might be available too, bit that's not bad – particularly if you're physically unable to move a mouse.) But should the mouse have one, two, or three buttons ? All these possibilities are used. The advantage of many buttons is that you don't have to resort to complicated codes like double-clicking to get enough different signals for effective control – but in systems modelled on the IBM Personal Computer, there are two buttons and double clicking is still used, with the second button notably underemployed. Even within the Macintosh system, double-clicking has (or had in earlier versions) at least two meanings. To the Finder, double-clicking an icon means "select and open", with "open" interpreted sensibly enough according to the nature of the icon; double-clicking the file name means "select the name for editing"; double-clicking the file name again (perhaps even without moving the pointer !) means "select this word", as it does in word processing software. The first and last meanings are not obviously related; the principle of consistency is at least bent. The "select for editing" meaning is just about consistent with the first interpretation, but, as it draws a distinction between file name and icon which is not usually evident, could itself be seen as an additional element of complexity. That's tidied up a bit in Version 7 of the system – so we have to unlearn the convention we had painfully worked out. Moral : designing conventions badly can be as bad as not having them.

A second important feature of the graphical interface is the menu bar. This is a quite elegant solution to the innate paradox of the modeless system : if you can't have modes, how can you run different programmes ? The answer is to retain the basic and consistent idea of selection, but to modify that by providing, in a consistent way, a set of things – actions, values, etc. – appropriate to the programme which can be selected. In this way, the variability inherent in running different programmes is exercised by the programmes themselves in offering different sets of things; the actions of someone using the programme remain consistent. But how do you use the menu bar ? When you come to something puzzling, you search the menu bar for some item which might be relevant, and try it. That only works if you can understand the items on the menu bar – and if the items are pictures rather than words that isn't always easy. The Macintosh menu has a pretty picture of the Apple apple at the left-hand end of its menu bar. There's nothing to indicate that it is a real menu item, not just a trade mark.

There's no real conclusion. Things are better than they used to be, and perhaps still slowly improving. Step by step we work out how to use our inventions to better advantage – then when we try to use the communal knowledge, we sue each other for breach of copyright.

IN THE LIMIT ....

If operating system developers really take these ideas seriously, will all the systems eventually converge to a universal operating system which is so elegant and simple that it can be used by anyone ? Perhaps not just yet, but there are certainly moves towards the universal operating system. So far, the front runner seems to be something or other strongly based on Unix. It is hard to avoid the suspicion that this choice has nothing to do with careful design, and everything to do with the accidental fact that Unix was one of the few operating systems which had actually been implemented on a very wide range of computers. It is certainly possible to argue persuasively that the Pick operating system would have been superior, particularly for commercial (which is to say, majority) use, but it didn't happen.

At the system level, there are rather more detailed proposals for universal standards. The best known is Posix (which we mentioned when looking to the future at the end of the *HISTORY* section), advocated by the IEEE (Institute of Electrical and Electronic Engineers, based in the USA but eager to be seen as an international body). Posix is based on Unix again, simply because variants of Unix are probably found on a wider variety of computers than any other system. In fact, Posix is mainly concerned with developing a universal system-level interface, which is comparatively easy; to develop a universal implementation which would really allow one to move things about at will from computer to computer is a lot harder.

REFERENCES.

REQ7 : R. Tagg, M. Sandford : "Where to now that the mouse has arrived ?", *Computer Bulletin Series 2* **#42**, 2 (December, 1984).

REQ8 : Jane Lawrence : "Common ground ?", *Computing* (8 December 1988).

REQ9 : R. Miles, *Computing* (4 July 1991).