PEOPLE TALKING TO COMPUTERS

WHAT DO WE NEED ?

The essential requirement is that we should be able to tell computer systems what we want them to do; but there are many ways in which this can be achieved. People's requirements also depend on their functions – so system programmers, software developers, and people just trying to run programmes to get work done have different requirements. In any case, though, we have two things to do : we have to describe the task to be performed, and we have to convey the description to the system.

DESCRIBING THE TASK.

Every task which is performed by a computer involves running some sort of programme, and in many cases it's also necessary to identify some collection of data to be used or to be constructed, so most instructions contains several parts. Typically, there is a verb, which says what has to be done, and a noun, which says to what the action should be applied; these are commonly represented by the names of the programme and the data files. There may also be qualifying clauses describing just how the action should be performed.

There is debate on which of these is the primary component. Traditional systems place the verb at the beginning of the instruction, implying that it is seen as the main component. In an instruction like **edit letter**, the **edit** component, identifying the action to be performed, comes first. This seems very natural to anyone who knows anything about how computers work : to get anything done, the first thing you have to do is to load the programme which is going to do it, and the programme will then need to know what to work on.

With the shift in the nature of the people who use computers from experts to the population at large, it might not be sensible to assume that everyone will find it natural to think in this way. In practice, people don't want to run programmes – they just want to get work done. What you really want to do is to work on your document, which happens to be called "letter". It's obvious to you that "letter" is a letter, and there's only one sort of thing you could sensibly want to do with it. You therefore find it natural to begin by selecting "letter" – and you might not even know about "edit". This is the *object-oriented* style, more common with WIMP interfaces – but really the style and the interface you use are independent. (Notice that an object-oriented interface style has no necessary connection with object-oriented systems design, which in turn need have little to do with object-oriented programming.)

The reasoning behind this view is based on the supposition that the way we use computers has changed. The shift is evident in the preceding two paragraphs : in the first, we regard the computer as a servant, and tell it what to do; in the second, we are intending to do the job ourselves, and regarding the computer almost as an incidental participant in the operation. If you want to alter a letter without using a computer, you don't give yourself an instruction like **edit letter**; you start by looking for the letter, because you *know* what you want to do with it, so the object-oriented view is appropriate.

There is another point of view which sidesteps this debate, and its not-very-convincing appeals to hypotheses on how we think about editing a letter. Go back to our statement of the operating system's job – "to produce results as instructed" – and consider its focus; it is on the result. What we want is an edited letter, and that defines a task, which is toeditaletter. That's the thing we have to define in order to produce the instruction – not the object, nor the action, but the task. Now, what's the easiest way to specify that task ? Eventually we have to define the two components, but we can take different routes. Here are three cases :

- If we are using a system in which neither of the components is able to offer any additional help, it doesn't matter which order we use. We have to do the work anyway. It's no harder to type `letter edit` than `edit letter`.

- If we have a helpful system which is showing us a representation (icon or menu item) of the letter, then we can select that first. We then have to identify the operation we want performed on the letter – but there are rather few operations which make sense : in practice, editing is almost always appropriate, and often assumed. In effect, the letter is helping us to complete the task definition.

- But what if we can't find the letter ? In this case, it might make sense to start with the editor, and to use that to explore the system in search of files which it can edit. This saves us from having to inspect all the files, many of which might simply be not editable at all. This is using the editor to help us to find the letter, and thereby again complete the task definition.

Which method do you use with a GUI system ? We find that we use both – sometimes in combination, first selecting any editable document as a quick way of finding the editor, then using the editor to explore for editable documents. That's what you might expect from the *task-oriented* style we've suggested.

In practice, both styles of computer use are important : sometimes we want an assistant, sometimes we want a slave, so it makes sense to have both styles of interface available. If you can only have one, though, using the traditional textual interface to start a text editor is easier than using a graphical interface to predefine a list of tasks to be done.

While we have the letter example in mind, it's interesting to follow it a little further, for we have not yet described the whole task. What do we do when we've completed our literary efforts, and the letter is ready to - well, to what ? We might want to save it for further attention later, or print it, or send it somewhere by electronic mail, or send it by facsimile. (Or several other things, but those are the alternatives immediately available to me at this moment, so this is a guaranteed real example !) This shows us two things, one general and one specific.

The general consideration is related to the question of instruction style which we mentioned earlier. The example shows that we don't always want to describe a task completely when we begin it; sometime we can give a complete description at the start, but in other cases it might be that events which occur during task will help us to decide what to do later on. In early systems, it was usually not possible to do this; a task had to be fully described when it was initiated, and any variation in procedure managed afterwards. To do better, we require additional system facilities, and in the particular case of the letter the facility we require is *device independence*. We mentioned device independence briefly in *ONWARDS AND UPWARDS – OPERATING SYSTEMS*, and will discuss it in more detail in *DEVICES*; in this chapter we're concerned with drawing up our list of requirements for the system, so we imagine it added to the list.

More specifically, we need some sort of instruction to tell the system what to do with our letter, and we want to be able to issue the instruction while we're using the editor. We must therefore be able to communicate with the system while we're doing something else; this is another entry in our list of requirements. Notice that we are listing requirements, not suggesting implementation techniques, of which there might be many. In the Macintosh system, we carry out the required communication fairly directly by leaving the editor for a while and using the Chooser; in Windows software, it is usually done through the editing software, typically through the File menu. The details are not important for the moment, but it is worth remarking that in early systems neither of these ways of communicating with the system was available.

ISSUING THE INSTRUCTION.

With either of the two approaches above, there are several ways in which the instruction can be conveyed to the system. Which you can use depends on the devices and media available, and on where the instructions originate. Whichever you use, the signals which you produce must eventually be converted into a standard set of signals comprehensible to the operating system, which will be something amounting to a set of values for internal variables which the system will use to determine what to do next. The interface is therefore in principle separable from the system (as with the Unix shell); it should be possible to change from one sort of interface to another quite simply. In practice, this isn't so in most cases, but that says more about the design of the system than the principles involved.

The instructions which the system obeys to do your work can originate directly from you, or remotely (in space or time) from you, or from a programme working on your behalf. The common expectation now is that you will be interacting directly with a computer as your work is carried out. In this environment, you can usually expect to use either textual instructions or (given a suitable high definition screen, plenty of memory, and a fastish processor – almost universal nowadays, but only because of phenomenal engineering feats in hardware development over the last twenty or so years) some sort of graphics interface in which you select from a set of possibilities offered by the computer. We shall discuss details later; for the moment, the important difference is that between you taking sole responsibility for the instructions (as you normally do if you type your input) and cooperation between you and the computer system (as in a menu or graphical interface, where you select an instruction or icon displayed by the system).

If you are working from a distance, cooperation might be difficult to achieve (if you are distant in space - ask anyone who uses the world-wide web over a slow telephone line) or impossible (if you are distant in time). Clearly, cooperative modes cannot easily be used if you're composing the instructions long before they are to be executed. That's just like ordinary computer programming, so you might expect that at least there would be some sort of syntax checker which could make sure that you didn't make silly mistakes, but they're not common. The earliest example of this sort of computer use is the coding form, and the same idea persists today if you write a *command file* (next chapter) containing instructions to be executed later.

If you are physically remote, the quality of the cooperation which can be achieved depends on the capacity of the communications channels which you are using. Technically, essentially the same interface could be presented anywhere in the world (though time delays can cause problems if you're much further away than that). When we first wrote these notes, such remote interaction was expensive and unusual; now it's commonplace, and people do it from home, mainly through the World-Wide Web, for fun. This is certainly a Great Leap; whether forwards, backwards, or sideways is a matter for debate.

> *Why "command file" ? Because the things we have been calling "instructions" are often called "commands". Why ? A command is a rather special exercise of will – you command an army, or you command a rebellious underling to get into line. You don't, if you're sane, command a machine to do things. You give an instruction – and if the machine doesn't cooperate, you get an engineer to mend it.*

If you want a programme to issue instructions on your behalf, then the situation is quite different. (– unless the programme issuing the instructions is running in a different computer from the system which is to obey them, in which case you write your programme to make your computer look like a terminal when seen from the other. For a long time, this was the major means of communication between programmes running on different machines, and was only really superseded when network protocols developed over the last twenty years or so.) Now you are, in effect, issuing instructions to the operating system from inside. For a very long time, it was impossible to do this with most operating systems, unless you were willing to patch the system code. Many people were very willing to do just that (if they could get hold of the code, which was another problem), but the result was that many different ways to link to the system were developed, which were usually incompatible. More recently, as systems have become more modular, it has been recognised that it's sensible to provide access to system facilities, and more systems are defining conventions to make this easier. The usual approach is to define an *application programmer interface* (API) which anyone can use, and which is properly specified to give uniform behaviour.

COMPARE :

Lane and Mooney[INT3], Chapter 4.

QUESTIONS.

What's the easiest way to describe a job which has to be done ? How does it depend on your knowledge of the job and of the system ?

Under what circumstances would each of the modes of communication mentioned be good ? – or bad ?

Interfaces using pens and voice communication are beginning to appear. How do they fit in with this scheme ?

How must the system be designed if it is to interpret the name of an object as implying some action on the object ?

What happens if you don't want to *edit* the letter, but want to *send* it to someone by electronic mail ?

Consider our remarks on how to give instructions to edit a letter. If you want to begin a new letter, there is no object to talk about except, perhaps, a blank sheet of paper, which could equally well be used to draw a picture or start some calculations. How should the object-oriented interface style work in this case ?

Consider all the input interface types mentioned in this chapter. In each case, what sort of computation must the interface software carry out to convert the input signals which it receives into the "standard set of signals comprehensible to the operating system" which we mentioned ?
