USER INTERFACE STYLES

(S.W.Draper , GIST, University of Glasgow. Feb 1996 disk Courses 6 file Styles.ho.12)

This handout reproduces printed material for a video course on HCI.

Contents

Introductory examples of style

Without going out of the way, it is common to come across quite a few different ways of interacting with computers. A commonsense list of interaction styles might be: command languages, push-buttons (function keys), direct manipulation, form filling, menu systems.

The UNIX system uses what used to be the most common way of interacting with a computer, sitting at a terminal or console and typing commands, which are then executed by the computer. These command language systems require quite a lot of learned knowledge: if someone asks for help, it often seems easier to do the task for them than to explain all the things they would need to know to do it themselves.

In contrast the push-button style common in ATMs (the cash dispensers used by banks — "Automatic Teller Machines") can usually be managed almost at once. The push-button style is common on many small appliances, but can also be used in some computer applications, where a special function key is provided for each operation.

On some computers there is the functional power and complexity of a system like the first, but with almost the simplicity of the interface of the second: the "direct manipulation" style of many drawing programs and WYSIWYG (What You See Is What You Get) editors. Central to this style is the extension of the direct push-button feel from function keys to the objects themselves (text, drawn figures, etc.).

Form-filling is a style where a number of separate choices are simultaneously presented, but can be made in any order. Finally, menu systems are in some ways the most common style. As we shall see, there are many variants, spanning the range between the other styles.

Some aspects of styles

The different interface styles mentioned above look very different even at first glance. But are these necessary differences, dictated by the job they are doing? Often you can see how the same job could be done in each of several interface styles. Thus most bank machines make the user type in the amount in pounds, using digits even though any amount that isn't a multiple of £5 will be disallowed: this is like a command language, using a standard keyboard and introducing the possibility of user errors. But it could have been done by special keys, one for each allowed amount. Similarly, a screen version of a pocket calculator lets you use either keys or the mouse to enter numbers. Other examples are typing "rm foo" in Unix vs. dragging a file icon to the trashcan in the Mac Finder; typing "vi" in Unix versus double clicking on the icon for an editor application; typing a line of Postscript to News vs. creating a square in Macdraw.

Thus "interaction style" means a constellation of standard solutions to the problem of doing input and output — the "look and feel" of an interface. The fact that there are often alternative solutions means that a designer can exercise choice, perhaps by paying attention to issues of consistency, smartness, distinctiveness, or "house style".

However the styles also differ in their direct effects on users: on the learning burden (how many little facts the user must learn), the degree of visibility (how much of the system and its state is made visible and kept visible at any moment), the degree of interactivity (whether or not the user is locked into performing a fixed sequence, or can choose what kind of thing to do next), and on whether the physical equipment is large enough or fast enough to support them. Perhaps because of this, when you look closely you find that styles are often mixed e.g. an ATM mostly requires push-button operation of a function, but the user has to type in a sequence of numbers to represent the amount of cash (instead of having a separate button for each common amount), and as a consequence there is a suddenly increased need for an undo or edit function for correcting typing errors in this sub-area of the user interface.

Thus there are at least two levels of style. At a fairly low level there are issues such as menus versus command languages. We shall look at some of these and see that there are many variants of each — in fact in the end one can blend into another. Just as with clothing styles, there is no finite fixed set of possibilities (even if only a few are currently widely sold). These styles differ from each other in their effects on the user; but the differences between similar methods are often small enough that for simplicity a designer just works with a few widely spaced alternatives.

At a higher level of the overall style of a program or family of programs, "style" is almost always a standardised set of interaction methods: e.g. a particular version of menus, plus one type of dialogue box and so on. Thus even in relatively uniform interfaces, styles in the sense of interaction techniques are usually highly mixed.

Windows

Although you may find styles closely mixed up (as in ATM digit keys, or a screen calculator that will accept either number keys or mouse clicks), they are usually separated either in time or space. Thus in DOS or Unix, you may type in a command (command language style) which summons up a program that uses direct manipulation style; or on the Mac, selecting a menu command may force you into a dialogue box, where you have no flexibility of choice, and have to use typing not the mouse (typically, commands for creating a file demand that the user type in the new name). Those are examples of separating styles in time. Still more common today is separating them by space, by dividing the screen into areas within which different styles are active. These areas are often called "windows", though terminology varies. On a small scale, a pulldown menu is a window within which only pointing is allowed, and where releasing the mouse button dismisses the whole display area (i.e. the menu or "window"). On a large scale, the term "window" refers to a large area within which whole programs often run. From the viewpoint of interaction styles, windows are islands of relatively uniform style, which however may differ greatly from each other in language and customs.

But more often windows are used as system, not user interface, constructs: not to mark out separate styles, but to mark separate processes which may or may not share a style, and to switch user input between them. Such windows have their own style, not only to distinguish them visually but because they have their own set of commands for moving, scrolling, sizing, switching input etc. However these window commands are not different in kind, or in the possible styles they might be expressed in, from other commands, and we shall not discuss them separately.

Command languages

In a command language, the user must type in a complete command (command name plus arguments) without any prompt other than a general one indicating readiness. There may or may not be a response: often silence and another general prompt is all the indication there is of success (or failure). It is not like natural language, where within wide limits anything may be said, but rather like commands over a voice pipe in a ship: though the units are "words", they are chosen from a special vocabulary. Thus in fact only a limited number of commands are allowed; it is just that the system does not give the user any information about them in advance. To get information, the user must ask for it explicitly. On the other hand, there is no delay while information is printed by the system and then read by the user: users can issue commands as fast as they can type. It is typical of such systems that they use short command names to maximise this speed. The command names in this style of editor are frequently a single letter, despite the problems this can lead to in remembering their meaning. Thus in general, command language styles maximise speed (usability for experts) at the expense of learning burden. They are often chosen because the display is small and slow relative to the size of the command set.

Menus

A common alternative is menu systems, where the finite possible set of commands is displayed in some way, and the user issues a command by reference to such a display. In fact, there is almost never enough screen space to display all the commands at once. For instance in the Mac system the permanent display has only a single horizontal menu called the menubar, and the user must click on one of its items to see more choices. If you look at the width of the menu displayed, you will see that it is wider than its title, and that therefore you could not display all of the menus at once. Thus the user must still perform some action to get the information they need. Indeed in some menu systems, called popup menus, NO information is displayed until requested (e.g. Sun windows, or the Smalltalk language). This minimises the cost in space, but the user must have learned what to do to get the display, and must take the time to perform that action, and wait for the display to be drawn. On small machines this can be annoying, and even on fast machines there are frequently keyboard alternatives, implying that at least some users find it is worth the extra learning to save the execution time. Note that it is not only the display time: to issue a menu command users must position the mouse pointer, and that takes time because they must look at the screen while aiming the pointer.

Menus do not have to be vertical lists of words. For instance the Tools menu in Hypercard is a pulldown menu of icons (pictures), arranged in a grid, and in many drawing programs there is a permanently displayed "palette" of icons representing operations. Tearoff menus combine these ideas: they are opened like pulldown menus, but can be "torn off" and left permanently open like a palette. Menus can combine text and pictures (icons and words). Popup menus are not only not permanently visible like palettes, their superior menu is not permanently visible, (i.e. unlike the "menubar" for pulldown menus). Note that which menu pops up may depend both on which button you press, and on where you click. Menus can also be round "pie menus".

The common feature of all menu systems is not that commands are made visible without user action, but that a command can only be issued by following a route to it, and hence that they have organised all the commands into some system such that there is always a route to getting the right command displayed. Thus the user gains access by making a series of choices. The user must remember how to make this sequence of choices, so there is still some learning burden. The groupings are designed to make these choices obvious, but research indicates that designers have very modest success at this: you can often see users of menu systems searching menus for the command they want. This is a very small cost on small systems such as an editor; but in a system with hundreds of commands such as Unix or a CAD/CAM system, this is a considerable problem. From the viewpoint of learning, the difference is between having to recall a single complete name and recognising how to make a sequence of forced choices.

Command language systems may in fact offer help based, like menus, on the fact that there is a fixed (if large) set of choices underneath. In some versions of the Unix shell, for instance, you can ask for the current set of alternatives after having typed some letters. This is in effect a keyboard summoned pop-up menu. It may be compared with using the keyboard instead of the mouse to select from a file menu in Macwrite or other systems offering keyboard shortcuts to menu items. Another facility is command completion: you type enough of the command to make it unique, and a special key will cause the system to complete it.

Menus as the universal intermediate style

Thus menu systems may be thought of as part of a range of facilities for displaying subsets of the available commands in response to user choices expressed from mouse or keyboard. We have seen how there are many variations in menu techniques, which can be ordered along various dimensions. In fact all interaction styles (command languages, direct manipulation, form filling, push buttons) can be organised along these dimensions, and menus span the whole range, with the other styles forming various special cases.

The first dimension concerns how much information is displayed to assist the user's choice. The extremes are when all or no commands are permanently displayed (as in a palette system or a pure command system). These extremes are in fact respectively equivalent to the push-button and the command language interface styles.

A second dimension is whether the user is allowed to make useless choices (i.e. whether the design of the interface creates a whole class of errors). In general in a command language many character sequences are meaningless and cause only an error. One can regard the keyboard as a set of push buttons, and ask what proportion of combinations are meaningful. The same question applies to menus, although usually only legal operations are offered (so the proportion is 100%). However legality is often context dependent, so sometimes the user issues an illegal menu command, just as in a command language. To prevent this, menu items may be dynamically removed or disabled. (This is called semantic feedback: an issue explored further in the next unit.) However there are problems with this. Thus the designer has three choices in a menu system: allow the user to issue illegal commands, as in a command language; remove the item from the menu when it is illegal, although this confuses users by making menus vary unexpectedly; or leave the item but disable it (e.g. they are greyed out on the Mac), which prevents the error, but does not stop users being sometimes puzzled and frustrated by seeing the item they want but not being able to invoke it. When we also consider that because of the terseness favoured in command languages, in fact a large proportion of possible command names are used, then it still seems reasonable to view them as on a continuum with menus and push button interfaces. (For instance in the command driven editor "vi", 23 of the 26 lower case letters are valid commands; 46 of the 52 alphabetic characters, and all but 6 of 32 punctuation characters are valid as commands. On the other hand, in my local Unix system, only about 6% of the possible 2-letter combinations of lower case alphabetic command names are in fact valid, and the proportion is still lower for longer command names.)

Further dimensions are concerned with whether the effect of an operation depends only on the operation specified, or also on other settings (e.g. user pre-setting of line width or font); and the related issue of whether sequential order is imposed on the user. For instance form filling interfaces are the same as the push button style, or equivalently as the palette type of menu (permanently visible, 2-D rather than linear graphic layout) as regards the earlier dimensions, but differ in allowing free order of user operations. We shall comment further on direct manipulation systems below; here we may note simply that they are attempts to extend something like the push-button feel to more complex interfaces. However many interfaces that claim to be direct manipulation in style rely on pulldown menus, and therefore do not strictly qualify since their operations are not permanently visible, but available only indirectly via the (learned) intermediate operation of opening the right menu. Thus in this way too, menus cover the intermediate ground between all the other styles.

Finally we should note that the issue of pictorial versus textual representation is an entirely independent one. Commands and menus themselves can be represented by either words or pictures, and selected using either mouse or keyboard (as when a menu is summoned by a keystroke or command completion invoked). Even a pure (non-menu, unprompted) command can be pictorial when it may be called gestural: for instance when in the Mac Finder the file deletion operation is performed by dragging the file icon to the wastebasket icon.

Arguments to commands

When it comes to arguments, rather than commands, the issues are basically the same. However since objects to be supplied as arguments may come from a set that changes (a dynamic set) (e.g. the files that happen to be on a particular floppy disk), the menus cannot be fixed by the designer. Scrolling menus are one answer (e.g. menus of available files), which are also sometimes used for large fixed sets of commands if these haven't been broken up further. These could in principle be pictorial.

However arguments to commands may not have to refer to an existing finite set of objects e.g. a name for a new file which may be anything, not a forced choice from a finite set. Then there seems to be no alternative to command language style input. Note however that facilities to help the user can still be provided: being able to edit the text typed; a greyed out scrolling menu to show other file names, both to avoid re-using an existing one unintentionally, and to show existing ones as models; a dialogue to choose the directory by menu selection; supplying the current name as text for editing. Such helpful facilities are directed not just at the choice of words to input but at hints about the meaning.

Another type of non-selected argument is pictorial: the endpoints for drawing a line or box. As with a new name, the number of possible values is too big to list as separate items. However instead of typing in numbers, the user may be asked to point. Scrollbars are a one-dimensional form of this: the user chooses a point in a fixed range. Scroll bars are a pictorial menu for a quantity. Alternatively, at least for theoretical purposes, one could regard the screen used by drawing programs as a 2-D menu of exhaustively listed alternative points: clicking by the mouse to specify where a point or line is to be drawn is in effect choosing one out of the alternative allowed points.

Complete commands: operation plus arguments

As the above shows, presenting arguments rather than operations to the user only extends the issues to be dealt with by menus to the extent that dynamically formed scrolling menus may be necessary, as are graphical input methods like sliders and scrollbars. Considering how to present sets of choices to form a complete, complex command raises other issues.

Both default values for arguments and more importantly the syntax of a whole command — number of arguments, and the fact that all must be supplied together and in order — lead to dialogue boxes or "form filling". These may use a combination of a menu of alternatives and free-text type-in. These forms allow the user freedom about the order of filling in their parts, whereas dialogue boxes — another solution to forcing the user to specify arguments — force a fixed order on the user inasmuch as you only get the box after selecting the command. (Comparable sequential constraints are often found in pictorial systems e.g. for drawing.) Some interfaces take this rigid order further: they ask a sequence of questions, one per argument, in a fixed order.

These example solutions point up two related issues of great importance in interface techniques. The first is that the effect or meaning of many user actions (e.g. issuing a menu command) depends on what other choices have been set. For instance in a word processor, the effect of a delete operation depends on what text has been selected; the effect of typing in text depends on what font size and style have been selected. Thus system actions depend on <u>combinations</u> of user inputs. A simple push button style does not directly solve this, unless it is possible to provide a separate button for every possible combination. Instead, combinations are used, and the issue of syntax becomes important — of what combinations are allowed, and whether the order of user input actions is significant.

This latter is the second issue: whether the <u>sequence</u> (order) of user actions is constrained. Just as most natural languages constrain order, but some such as Latin are said to be free order (within the unit of a sentence), so most but not all interface styles constrain order. Thus the Unix shell languages are verb-noun (command then arguments), as are many drawing programs (select a tool then specify the positions it takes as arguments); while editors are often noun-verb (select a position or text, then select the operation such as insert, delete, or change font). However form filling is a salient example of free-order input, and it is possible though very unusual to have "intelligent" commands or command languages that analyse the types of the tokens and infer what roles each token is to play. (For instance, a copy command could examine its arguments and use their attributes — whether they exist, and whether they have read or write permissions set — to work out, in many cases, which was the source and which the destination. Similarly the fact that the token for "copy" was executable and the other tokens not, could show that it was the command and the others must be its arguments.)

In menus, items may be mutually either inclusive (as in font style attributes such as bold, superscript) or exclusive (as in commands) — as pointed out systematically in Apperley & Spence (1989). This is another aspect of designing the combinatory aspect of commands, as is also the division of expression elements into verbs and nouns. One can make "delete" a verb as in Unix "rm foo", or a noun as in the Mac where deletion is represented by the trashcan together with a generic verb of dragging (cf. "do the action of deletion on 'foo'"). This design freedom can be used to allow free-order syntax. If each element is typed, then the system can remember the last selection of each type (say, "delete" and "copy" might be actions, files and folders might be objects). Selections of any type can be done in any order: the system just remembers the last of each type. Whenever the verb "do" is selected (cf. the "OK" or completion button in a form or dialogue box), then the system executes a command made out of the remembered selections.

Reprise

Many initial impressions of style are based on things like whether the system uses a mouse or has pictorial representations. Underlying these features are the questions of what constraints are actually present, and how these are presented to the user; what information the user needs, and whether this is presented. The different cases tend to mean that different styles are more or less necessary for them: you cannot sensibly have a menu of all the possible names for a new file. On the other hand, almost all designs present less information than is needed by users because of resource constraints. For instance only in very small systems can all the commands be permanently displayed.

Above we began by looking at methods for presenting commands, which are items from a fixed set. Extending this to dynamic sets, such as files, covers one type of argument. Another type requires more free-form input. The syntax of a command (the number of arguments, their type, their relation to each other) also needs expressing. Form-filling is a general approach that gives the user the information they need, while allowing considerable freedom in the order of input. Less flexible methods have a rigid order. Less informative techniques demand that the user know what to type in rather than prompting for it. These are independent of each other: it is possible to have a system that does not prompt for arguments, but allows them to be given in any order. Thus users need not remember the order, but only the number and type. (This could be billed as an intelligent system: certainly it requires a surprising amount of code to search over all possible meanings for the arguments and pick the most plausible mapping.)

Examples: applying the framework

This section reviews some other examples to illustrate the framework developed so far.

The Mac finder presents files as icons. These in effect constitute a 2-D pictorial menu whose layout is user-defined. (The other difference is that in many menus, all pointer positions select the nearest item, whereas in the finder you can point between icons and fail to select any of them.) Note too that the menubar constitutes a menu of menus; and that its first item is pictorial (the apple symbol), whereas the rest are textual.

The text editor or word processor Macwrite, like many other similar applications, exhibits pulldown menus for commands (so as to leave as much screen as possible for the main display of text), and a scrollbar that might be thought of as a 1-D continuous valued menu of position. The commands to do with text have both their arguments and their effects immediately mirrored in the text display, and so qualify as a direct manipulation design except that the commands themselves are hidden in the menus. However the commands to do with file manipulation (e.g. saving the text to disk) are indirect. Files are not represented by screen objects until after a file command is invoked. In the case of the save command there is no feedback except noise from the disk drive: as in Unix, the only confirmation of successful execution is the absence of an error message. On the other hand the Save As command shows a very subtle touch. This command is for re-naming the current file. Since a new name is required, there is no alternative to demanding that the user type in the name: ideas about direct manipulation offer no help for creating new objects which obviously cannot be displayed until created. The designers however have supplied the information that the user in fact may well require while choosing a new name: a list of names already taken by existing files, in the form of a scrolling menu of files which can be read but not of course selected. This is a novel application of a menu (displaying information, not offering choice), and also illustrates the deep principle of analysing what information the user needs and supplying it when they need it.

CADCAM systems are typically attempts at direct manipulation systems, compromised even more than text editors by the fact that their huge collection of commands necessitates a large hierarchical menu system in which finding a function is a serious problem, often requiring prior learning or elaborate help systems.

The first thing to note about spreadsheets is that they are fundamentally different from the usual sense of direct manipulation, which is to display the effects or state of the program and to achieve new state by directly editing the current state (e.g. type in the text you want, reposition part of a drawing). In spreadsheets, the user knows the formula or calculation they want, not the result. In this respect they are like much database use: the user knows a description or specification, and the whole point of the program is to calculate its value in a particular case. Given this basic goal, spreadsheets are an improvement over programming languages because they not only show the results and keep them constantly up to date, they also show all the intermediate results (something which at best you only get after a long interaction with the debugger in most programming languages). However, since to date most spreadsheets show only these results, they are also indirect in that they do not show the formulae which are what the user needs to manipulate. Another feature to note, at least in some, is the large, single, scrolling menu for formulae, rather than a structured hierarchical one.

Caveat: the limited generality of the discussion so far

We began with a casual list of what seem obviously different styles of user interface, and drew various conclu-sions about their underlying similarities and differences. One kind of (cognitive or psychological) property con-cerns their effects on the user: issues of usability, learnability, and the amount of useful information displayed. Other kinds of property are technical (computer science) ones: whether sequential constraints are imposed on the user, whether or not user actions depend for their effect on combinations of inputs. These latter issues are expl-ored in greater depth in the next unit, which also makes it clear that they apply repeatedly at a number of levels.

Take, for example, the issue of constraints on sequence. Within most forms of dialogue boxes, the user is free to fill in the options in any order, but cannot leave the box temporarily to do something else: free order at one level, but forced sequence at the outer level. Again, at the lowest level of many interfaces if you use the mouse you may move it and press its buttons in any order; but on a keyboard, you may not press alphabetic keys simultaneously, and even when two keys may be pressed at once (e.g. <shift> and 'A') only one order is meaningful.

Before going on to conclusions about how these properties make up the issue called "style", we should be clear about how little has been covered. Firstly, the interaction techniques we have mentioned have all been at one level, and other levels are also important. Next, the kind of interaction technique we have looked at will probably eventually be seen as occupying only one corner of the space of possible techniques: other possibilities such as "intelligent" interfaces have not been mentioned.

Non-direct and "intelligent" interface techniques

The techniques discussed above vary in the amount of displayed information they offer the user before demanding a choice, but they are all basically concerned with having the user specify exactly what actions are to be taken, and to preventing anything other than a clear and complete specification being issued by the user and accepted by the system. It is not hard to imagine (but it is hard to point to familiar examples of) rather different kinds of interaction. For instance in ordinary life (though not in, say, military situations) it is common to issue rather vague directives, and expect the recipient to detect and discuss any problems with it while filling in unspecified but unproblematic parts. There are many mechanisms in human conversation that support this: inferring what is implied or intended by an utterance, correcting errors in the other's speech, asking for clarifications when necessary, not merely objecting to errors but helpfully stating exactly where some utterance or request breaks down (presupposition diagnosis). All of these allow the speaker to go far beyond what they are sure of being correct, by relying on the hearer's active cooperation in any repairs that might be needed. This greatly reduces the amount of accurate knowledge needed by the speaker in advance. These conversational abilities are nothing to do with the actual syntax and vocabulary of the language: they could be implemented in the context of simple command languages (say), and need not be associated with natural language interfaces. Such interfaces might be called "intelligent" on the grounds that they are complex to implement, and mimic certain human abilities. What is certain is that they are in an important sense the opposite of direct (manipulation) interfaces. Users rely on the system to interpret and debug their requests, instead of being responsible for issuing detailed and exact commands.

Styles of output

Another way in which all the above concerns only a fraction of the space of user interface design is that all the "interaction styles" discussed above are really techniques for organising user input: they all end by issuing a command to the system. Simple symmetry reminds us to expect an equal number of issues in treating system output from programs. This concentration on user input techniques may be seen as a reflection of the major impact of user interfaces at the moment. The main difference from the batch computing of the 1960s is represented by the personal computer: users expect to interact personally and directly. While computer output does not look so different from what it was (and while so many applications are oriented around printing onto paper — WYSIWYG usually turns out to mean that what you see on the screen is what will be printed on paper) it is user input which has changed, and which dominates a user's impression of what it is to use a computer. In the 1990s, however, the emphasis is changing to multi-media and to the production of output of a qualitatively different kind. The emphasis of HCI may be shifting to output styles.

The issue is actually of course already present. A simple example is the question of how to present text in an editor on the screen. Good editors often offer not only a representation of the printed page, but another one showing "white space" characters explicitly. Less common, but arguably useful, is a representation that ignores printing but is convenient for editing the content e.g. without page boundaries. These are issues of output "style": how a program should present its main content. Data visualisation is a whole field devoted to aspects of this: to how to present complex data to users. One can view all computer graphics as a subfield of output style. The design issue is basically the question of how, given a program, to present its results to the user. Of course much of this is not specific to computers — for instance Tufte's (1983) book on how to design graphical presentations is primarily about printed representations, but can equally be applied to graphs etc. on a computer screen. Similarly much work on multi-media in computer systems can be guided by existing design and/or human factors expertise in those media (e.g. animation, sound both in art and in the design of control panels for process control).

Effects of commands

There is another type of information we have not so far discussed: feedback information on the effect of a command execution. Again, methods differ in how much information they give the user, with the less informative ones relying on the user having learned special skills. A given system or command may give no feedback — typically the user must then either follow up a command with another to inspect the result or trust to luck. This is notoriously so in basic Unix, leading to standard patterns of commands like **cd** then **pwd** (change directory, then print the current directory to check where you are), **rm** and then **ls** (remove a file, then list files to see if it has disappeared); it is also true of the **save** command in many Mac applications: the only confirmation that it has worked is the sound from the disk drive. A system may give error messages (so that silence indicates success); or printed output messages (e.g. DOS **copy** command); or dynamically maintained state displays whose change shows the effect of any command, as in any WYSIWYG editor where the visible change of text on the screen is confirmation of the effect of commands.

Thus not all output is to express the main results of the program. Some is to support user input e.g. displaying menus, and its third major function is feedback on commands. In summary, feedback may be divided into that representing state, affordance, or the success of a command. At the top level, representing state is often the same thing as representing the main output of a program e.g. the text in a word processor. However there are many other kinds of state to represent e.g. which items are selected, which mode the system is in. "Affordance" means the potential for action by the user: showing what buttons may be clicked, what objects are available for operating on, which menu items are disabled. Feedback on the success of a command may not always be necessary e.g. if the command is instantly reflected in a change of state. However this category includes status and progress indicators, error messages, and so on.

Inter-Referential IO

Finally, besides user input and system output, there is still another issue of how they do (or do not) interact and cross-refer. What may be termed "inter-referential IO" is the issue of allowing input and output to refer to other input and output. Examples include being able to copy part of the output and use it as an input command (input referring to output), or being able to click on an error message and ask for an expanded explanation (making later output refer to earlier output), or the error message literally pointing (e.g. using a thin line drawn on the screen) to the text it is complaining about (again, output to output reference). See Draper (1986) in the course textbook for an extended discussion.

The direct manipulation approach may be thought of as attempting to match input and output styles, so that the program output is an effective and perceptually successful ("direct") representation of the domain (e.g. text, 3D geometric solids, etc.) and also so that user input is organised through that representation e.g. by being able to select and modify that representation directly. Such an attempt to match and balance input and output issues is part of this third aspect of inter-referential IO. It seeks to link input and output by a common representation so that the output display that shows the internal state of the program is also the prompt for, and means of, user input. Thus not only might much more be said about the organisation of user input techniques, but there is a similar amount to be studied about system output style, and inter-referential IO.

Summary

We have looked at interaction styles for user input, although a fuller treatment would spend equal time on output styles, on inter-referential IO or equal opportunity interactivity (Thimbleby 1990, ch.15), and on how each of these issues applies at several levels. We have concentrated on "direct" systems, where commands are predictable and conversely the user is expected to be explicit and exact about what they want the system to do (and so must know a lot about what it does), even though other kinds of "intelligent" interaction are easily conceivable.

In looking at interaction styles, we have identified two kinds of underlying factors: technical (computer science) aspects, and cognitive (user-oriented, psychological) ones (see fig.3). Among the former (taken up again the next unit) are: combinations, sequence constraints, multiple levels at which these issues apply.

Among cognitive issues, the tradeoff between two factors governs the issue of user interface style or interaction technique: the <u>learning burden</u> for users with less than total knowledge, and the <u>cost of execution</u> given the necessary knowledge ("learnability" and "usability"). This tradeoff applies to many different kinds of information: command names, syntax, the function (effect) of a given command, the effects of a particular command execution, the existence of variable sets of objects such as files.

The obvious way to escape these pincers is <u>visibility</u>: displaying the information on the screen. This means that it is there (so no learning burden) but does not slow up the knowledgeable user. However this escape route is strictly limited in capacity by the hardware of the interface: if display is slow then user execution is penalised by waiting for the display (this is the main reason why display editors and menus were seldom seen when interaction was by means of 10 character per second terminals), and whatever its speed, there is limited screen space so only part of the possible information can be shown at any one time.

All the existing and possible styles can be seen as different solutions to this tradeoff between learnability and usability for a given "budget" made up of the size of the application (which determines the total amount of information concerned) and the time and space capacity of the display technology. A major source of difference between interaction styles is on which type or types of information they choose to spend the budget.

Interactivity — the degree of freedom allowed the user in the choice of their next action, i.e. how much their actions are constrained to certain sequential chains — is largely a consequence of choices about this tradeoff. In order to reduce the burden of learning the syntax of what arguments must be given with a command, some systems lead the user through a rigid series of prompts. This avoids the need to learn the syntax, but only works if the user already has enough information in their mind to fill in the answers demanded (the semantics required): subdialogues to consult other information are ruled out by some such styles, though form-filling maximises the flexibility in the order of answering questions. However interactivity is also desirable independently to support users swapping between tasks for unrelated external reasons (e.g. someone comes in and asks for something else, thus interrupting the user's current task).

Finally, an important issue in designing interaction styles, especially combinations of techniques, is whether they latch on to existing (motor) skills. This is both important and hard to predict. For instance if you place a truly naive user in front of a computer with a mouse, they will be unable to figure out how to use it at first; yet after only a few minutes training, they will find it "natural" that moving a device on a horizontal plane to one side should move a cursor in a vertical plane. Rotating the mouse 90 degrees will disrupt this ability: only some mappings seem to fit our existing skills. Buxton (e.g. 1986) repeatedly draws attention to how so far computer designers have almost entirely neglected human skills for coordinating limbs e.g. every car driver soon learns to use feet and hands together in precise control movements.

Another aspect of this latching on to existing human skills lies behind the widespread use of scrolling. No editor or word processor that allows multi-page documents is strictly speaking a direct manipulation system, since most of the document is not visible. However, people find it very easy to grasp that the window can be moved over the document to view any part they like. When you contrast this with how the major problem in hypertext systems is "navigation" i.e. the inability of users to find their way reliably around even very small systems, then you can begin to appreciate that scrolling is another case of a design latching on to a prior human skill so completely that it is hard to see that there ever was a problem. 3-D graphics workstations that allow the user to rotate and move 3-D depictions (e.g. of molecular structures, or engineering drawings) can be though of as generalised scrolling. The current interest in virtual realities is the latest move in this area. It is driven by the intuition that if computers can present a program's state via a metaphor of 3-D geometry, motion, and human manipulation via "hands", then this will be very powerful. No doubt this is roughly correct, but the examples of scrolling and the mouse suggest that literal imitation of reality may not be necessary to latch onto the relevant human skills. For a more than usually thoughtful commentary, see Mercurio & Erickson (1990).

Conclusion: Senses of "style"

Finally, what can be said about the issues lying behind the vague word "style"? In the text above we dealt at length with a particular range of input interaction styles that are currently common, showing how the actual design space is larger, and ending with an emphasis on the cognitive as opposed to the technical issues (since the latter are taken further in other units). In this concluding section we broaden the discussion by reviewing some other kinds of issue associated with the word "style".

Interactive input techniques

First the interpretation we have concentrated on in this unit is that of style in the sense of interaction technique, especially of user input technique. The different styles differ in their resulting usability and learnability. These differences seem to stem largely from three issues: whether user actions are more or less constrained into a fixed sequence; whether (how much) the information relevant to the user's choice of action is displayed when the choice must be made; and the extent to which illegal actions are permitted (user errors remain possible).

Metaphor

A different sense of "style" is that of metaphor. Metaphor on the whole does not have much impact on "style" as we have discussed it here, though it may be very important to the overall subjective look and feel. Metaphor refers to systems for improving the guessability of a system by referring to its parts by the names of entities in some other world: most famously, the desktop metaphor calls files "documents", the screen the "desktop", deletion "putting it in the wastebasket" etc. This could be done in a command language just as well as in a mouse and icon interface.

There has been considerable discussion in the research literature, some of which argues against the use of metaphors (Halasz & Moran 1982). One view of the issue consistent with the evidence so far, is that designers often find metaphors very useful in suggesting new yet consistent parts of their design, but they cannot be relied on to support users in guessing something effortlessly first time. Perhaps metaphors are mainly useful as mnemonics: for organising what you already know and are trying to memorise for reliable recall. That is why they are helpful to designers (organising many aspects of an interface), and sometimes also to users who have been shown the features; but not to first time users who have to guess features without other help.

Pictorial vs. textual

A lot of attention is sometimes paid to whether interfaces are graphical (i.e. pictorial) or textual e.g. there is considerable research on visual languages. Sometimes the hope is that if the interface relies on pictorial tokens (icons) then it will be comprehensible to people of all cultures. For domains that are already naturally pictorial e.g. engineering drawing, this is already true. For other domains, more or less arbitrary symbols must be learned: in other words a new language. Although one way to tackle language barriers is to advocate that everyone use a new international language like Esperanto, this does not avoid the need for learning. Every technique discussed in this unit can be applied to either pictorial or textual tokens: the issue of which to favour is independent of the other issues discussed.

Direct manipulation and its alternatives

The term "direct manipulation" is widely thought of as a style on a broad scale of the whole interface to a sizeable program, and we shall discuss that briefly here, whereas above it was mainly discussed in the context of alternative choices of style on a smaller scale within part of an interface. People do not agree about its exact meaning, as is apparent from the series of interesting but conflicting attempts to analyse it. Of course it scarcely matters what it "really" means: what matters is whether any of the ideas that emerge from the analyses tell us something interesting about what characterises good interfaces. Some of the analyses, however, come close to confounding "good" and "direct". A question to ask yourself is how could an interface be good but not be direct manipulation (under some definition) i.e. what are the alternative styles at this larger scale?

Shneiderman (1982) coined the term, and defined it in terms of: continuous representation of the object of interest; rapid, incremental, reversible operations with immediate feedback; and avoiding complex syntax. Basically, this is an approach in terms of piecewise features of technique, each independently likely to favour a good interface. Later, people came to feel that there was something more unitary about the feel of "real" direct manipulation designs, including something in the word "direct".

Laurel (1986) discusses "direct engagement" as the quality that good theatre and good interfaces share: the audience forgets the artificial nature of the interface and indeed its presence, and instead feels themselves to be directly in the arena portrayed. This suspension of disbelief, the forgetting of the tedious details between you and the "real" subject, is an essential quality of many good interfaces, and its absence is what is wrong with many bad ones. Laurel's is also an interesting approach because it is defined in terms of the quality of the user's experience, instead of (like Shneiderman) in terms of features of the implementation.

Many computer games which do succeed in gripping their users make a point of forcing the user to puzzle out the rules and commands: this discovery challenge is part of the enjoyment, and not a breakdown in engagement. Direct presentation of the available actions and effects in the represented domain is thus not a prerequisite of engagement. In fact direct engagement is not a property only of direct manipulation designs: in a theatre or film the audience is passively engaged, while chatting friends are actively engaged as participants in an activity but without specific goals; in a business meeting the participants are engaged as responsible co-directors i.e. the overall goal is jointly held but responsibility is shared, while finally in direct manipulation systems the user is both active and in sole control — solely responsible for both the goal and carrying it out. Thus direct manipulation is only one kind (style) of direct engagement, and other interface designs might aim at others.

As argued in this text, direct manipulation in the sense of the style available in many programs today comes from extending the direct, push-button feel of function keys to the objects of the operations themselves (text, figures, etc.). This is done by designing around the (output) representation of the main set of objects (e.g. the text of a document), and designing the user input technique to work through that representation e.g. by being able to select and modify that representation directly. This simultaneously matches input and output representations (less learning for the user), gives continuous feedback about the program's state and hence also for all the operations that change it, gives appropriate prompts for user action (in theory, you can regard the display as a menu of objects — e.g. words in the text — that may be selected for action), and satisfies the desire for "directness" by ensuring that anything you can see, you can change by doing something to its representation.

As noted earlier, many systems of this type in practice fall short of the ideal as just described. Their commands may not be permanently visible on palettes but hidden in pulldown menus, and many of the objects to be operated on may also be invisible by being outside the current range of a scrolling window. Thus what we have is a recipe for a style that is often successful in pleasing users (including directly engaging them), but we can be less sure about how important the exact fulfilment of some of the specification is. However we can certainly imagine quite different but desirable styles, so it is not an empty definition of "good": for instance "conversational" designs where users rely on the system to interpret and debug their requests, instead of being responsible for issuing detailed and exact commands.

One of the best known analyses of direct manipulation is that of Hutchins et al. (1986). They introduced an analysis of "directness" in terms of semantic and articulatory aspects, and of gulfs of execution and evaluation. This is valuable for analysing what is unsatisfactory in various designs. However it is hard to imagine an interface that is good yet indirect in their terms: their analysis seems rather to be of necessary properties for adequate design, than of what distinguishes a direct manipulation style.

The place where this surfaces most clearly is in considering interfaces that deal essentially with intensional descriptions: programming languages, spreadsheets, database retrieval. In all these cases, what the user has is a description and the point of the program is to compute what that description turns out to refer to. This seems fundamentally different from what direct manipulation is about: a relatively small set of known objects which the user points to and selects extensionally by recognition (not description). The Hutchins et al. analysis applies equally and usefully to the former case: for instance, it is important in database retrieval interfaces to reduce the semantic gulf by matching the retrieval language to the terms in which the user thinks. These ideas of gulfs do not however seem to allow for another kind of indirectness: that of specifying descriptions without already knowing exactly the results you expect; yet this is intrinsic to some tasks, and hence to user interfaces that support them. In database retrieval, for example, you cannot use the output representation (of an ennumerated set of items) as the input (a description of that set): the task is in effect defined as one which converts descriptions (from specification to results, description to ennumeration). Thus direct manipulation is not a candidate for these tasks, whereas the "intelligent" or "conversational" styles mentioned earlier are.

Visual design and style

Another connotation of "style" is that of styling, and more generally of visual design. Although perhaps it may be, or become, of central importance to HCI, historically it belongs to a professional (and educational) tradition very different from software engineering, including such relevant specialisations as product design and graphic information design (with emphases respectively on 3-D and 2-D issues). There seem to be two kinds of thing that visual designers might contribute to user interface design: achieving new kinds of goal, or achieving similar goals by a different approach. Examples of new additional goals (which could use remaining free dimensions of choice in the design) might be aesthetic ends (the colour of the keyboard, to pick an extreme example). Alternatively goals of the same functional kind as designers from a computing background might be achieved but to a higher degree by using not an overt analytic methodology but a skill derived from years of expert artisan training. (This is not to say that visual designers never use an explicit method: but they do not attempt to codify their practice to anything like the extent attempted in, say, exhaustive top down structured programming methodology.) Let us look briefly at each of these.

We noted earlier that there are sometimes free choices of interaction technique available. In fact, there are many more degrees of design freedom, as the following argument derived from Hofstadter shows. When people learn to read, they learn to recognise the letters of the alphabet printed in many different typefaces. Learning what counts as a letter 'A' regardless of style is a functional requirement of learning to read. However if you show people several different versions of 'A' and ask which best fits the style of a given version of 'B', they will show a considerable degree of consensus. This is remarkable, since it implies that we also learn to group letters by similarities of typographic style even though there is no obvious functional requirement for developing this skill. Thus without any special learning, people in fact are sensitive to dimensions of visual design that are independent of the ones fixed by the obvious functional design. For instance, you can often recognise a computer or a software package from a fuzzy glimpse of part of a screen in the background of a scene from a television programme — even though this recognition ability has never been important to you in using computers.

This argument may also have an application to complex sounds, and hence potentially to the design of auditory icons. Although it would seem that we cannot articulate very well about how a machine sounds, nor predict it, we nevertheless have a great capacity to learn what a sound means and to notice small changes in it. It is somewhat like our ability at perceiving human faces. It seems to be effortless, unconscious learning with considerable variation between individuals, not dependent upon direct functional need. People in fact come to use very subtle sound variations in their homes, their cars, and in other machinery they tend. Gaver is exploring the construction of auditory icons that draw upon everyday sounds, especially relatively universal invariants (e.g. distance is encoded in pitch and loudness). In fact, our learning ability and its successful application to industrial machinery by operators may mean that apparently artificial noises may do just as well as long as users are given time to learn. Such sounds may also be enjoyable and even thrilling, at least to some, despite being artificial, as steam locomotives show.

An indication of the enormous scope for this stylistic variation in user interfaces, at least in the visual domain, comes from comparing the current styles with those used in other machines that are logically equivalent from the point of view of the controls. Most controls whether in a word processor, a steam engine, a cooker, or a VCR consist of a some combination of individual on-off switches, selection between alternatives, or variable settings. These can be presented by buttons, menus, and sliders in today's computer interfaces; but in other domains by up-down switches, multi-position rotary switches or levers, and hand-wheels. It would be easy to present these graphically in an interface to give different sets of controls entirely different appearances.

One thing designers can do is to exploit this human ability by coordinating such "free variables" in the design of an interface. They might be used for decoration, fashion, "house style" (i.e. to make all the products of a company look like each other and unlike other products), or for deliberate variation so that, like the use of more than one font style in a single document, the differences allow the user to distinguish the different programs at a glance. Here the suggestion is that such dimensions, once identified, might after all be turned to functional uses, albeit for user tasks (like recognising a program) and time savings ("at a glance") that seem secondary to the main requirements. These design aims would also include traditional ones from visual design in other areas: concerns such as readability, good layout, and, most fundamentally, an attempt to make the artifact's essential structure directly apparent to the user.

Functional justifications of such design aims include: house style (recognising the manufacturer of an application), recognising the type of a message by its style (e.g. indicator, warning, error, fatal error, ...), allowing faster reading of a message, allowing faster extraction of relevant information from the screen, making important information more salient than the rest. A common argument is that by making style uniform i.e. "consistent", you save the user the burden of learning and guessing how to do things. This can be important: if you see a user familiar with, say, the Mac trying to do even simple editing on, say, the Smalltalk system, which has different conventions, then you realise that in today's designs there are many low level conventions that have to be learned, and by imposing a single such convention you save the user a lot of trouble. On the other hand, consider the huge range of different designs for door handles in rooms, many of which look quite different from each other. Only occasionally do they give trouble for a new user, so it seems that things can be designed to be self-evident i.e. guessable without relying on consistency (partly by exploiting constraints from the context: what apart from a handle do you expect to see on a door?). Thus a consistent style may not be necessary after all, or at least might be used for some other function than to ensure first time usability. Hence even within the issue of functional aims, design skills can take us far beyond crude guidelines such as "be consistent".

Besides expanding the kinds of functional design goal to be considered, considering traditional non-computer design also suggests ideas about the considerations that may be important in how people react to a design. One such is that of aesthetics. One traditional view is that there are a small number of eternal aesthetic dimensions that apply universally. This philosophical field now applies to HCI, and will be important as soon as gross aspects of function are served adequately by competing products. An alternative view is based on the observation of how public taste changes, and seems to be led by the artistic, advertising, and design communities. We could treat this as an effect of how familiarity (we see things as looking LIKE something) and recognisability are in themselves qualities, but clearly history-dependent ones. Alternatively we can treat it as fashion (being fashionable is the point i.e. the user goal), or as having a hidden logic of aesthetics, which artists are researching and the rest of us slowly follow on. The general suggestion here is that these are new kinds of user characteristic which need to be explored and matched for a more completely user-centered design.

Quite apart from the expansion of design goals, however, there remains the issue that to juggle large numbers of such design aims and to find a good solution requires a skill almost certainly forever beyond explicit analytic methodologies. It is however familiar in the design disciplines such as architecture, product design, and graphic design. A naive description of this skill might use terms such as "genius" or "natural talent"; a more sober view would ascribe it to the expertise derived from long training and practice; and cognitive science might think of this in turn in terms of connectionist models that can, given enough training, produce optimised solutions without any representation of analytic concepts or formal reasoning to support the skill. Thus visual design represents not only an area of knowledge, but a type of skill — an approach to the activity of design — which may turn out to be vital to user interface design practice. It may also mean that "interface style" comes to connote an attention to detailed design of a kind seldom discussed in HCI up to now.

Bibliography

M.D. Apperly & R. Spence (1989) "Lean Cuisine: a low fat notation for menus" Interacting with computers vol.1 pp.43-68.

W. Buxton (1986) "Interface as mimesis" ch.15 pp.319-337 of Norman & Draper (1986).

S.W. Draper (1986) "Display managers as the basis for user-machine communication" ch.4 pp.339-352 of Norman & Draper (1986).

F.G. Halasz & T.P. Moran (1982) "Analogy considered harmful" in Proc. CHI'82: Human factors in computing systems pp.15-17. (ACM: New York).

E.L. Hutchins, J.D. Hollan, & D.A. Norman (1986) "Direct Manipulation Interfaces" ch.5 pp.86-124 of Norman & Draper (1986).

B.K. Laurel (1986) "Interface as mimesis" ch.4 pp.67-85 of Norman & Draper (1986).

P.J. Mercurio & T.D. Erickson (1990) "Interactive scientific visualization: an assessment of a virtual reality system" pp.741-745 in Human Computer Interaction: INTERACT '90 eds. D. Diaper, D. Gilmore, G. Cockton, B. Shackel (North-Holland: Oxford).

D.A. Norman (1988) "The psychology of everyday things" (Basic books: New York).

D.A. Norman & S.W. Draper (eds.) (1986) "User centered system design" (Erlbaum: London).

B. Shneiderman (1982) "The future of interactive systems and the emergence of direct manipulation" Behavior and information technology vol.1 pp.237-256.

H. Thimbleby (1990) User Interface Design (Addison-Wesley: Wokingham).

E.R. Tufte (1983) The visual display of quantitative information (Graphics press: Cheshire, Connecticut).