Design of Interactive Systems

Harold Thimbleby,
Stirling University,
Scotland, FK9 4LA.

Email: harold@uk.ac.stir.cs
Phone: +44 786 73171

October 1, 1998

Abstract

This chapter presents an overview of major issues and techniques in user interface design.

To appear in,
J. A. McDermid, ed. *The Software Engineer's Reference Book*, Butterworths, 1990.

1 Introduction

It would be a truism to say that no computer could be used without a user interface, yet the user interface is often an afterthought in the design process: it is tagged on at the end, and many opportunities are missed. It has been said, and not in jest, that many user interfaces are outgrowths of the debugging tools built into the prototype program.

Problems start to become apparent after systems have been shipped and the first real use of them is being made. By then it is often too late to change early design commitments, and the best that can be done is to palliate the problem—or offer the user the expense of an alternative system. Various studies have been made of the cost of rectifying design errors in the user interface, and estimates put it around 60% of total software development cost. Of course, the end users also have consequential costs that this '60%' does not measure, arising from being trapped into using systems that are not as good as they could be. It has been repeatedly shown that simple improvements to many user interface designs result in increases in productivity and decreases in error rates—and leading to reduced training time, greater job satisfaction and reduced staff turnover.

Overview

This chapter is arranged to be read from start to finish. It starts with an overview—confirming that user interface design is hard—then examines design choices and principles. The principles do not so much answer questions ("should we include this?") so much as raise important issues. Design, however, is not just a matter of thinking about principles, but having methods to put them in to practice. The chapter therefore describes various design tools, evaluation methods and general approaches. The chapter concludes with a case study: this may be taken either as an example or as an exercise.

2 It's harder than you think

The Three Laws of Interactive Systems Design are

1. Know the user
2. Know the task
3. Allow for design errors—don't assume you can follow rules 1 and 2.

Rules 1 and 2 are often summarised as 'use the user's model,' the **user's model** being the user's 'program' that determines what they do and how they interpret their actions. The user's model, then, is normally established *before* any contact with a computer system, and the designer should go to some lengths to find out how users perform their activities—including error recovery and concealment!—prior to computerisation.

Rule 3 arises for two reasons: first, you can never really know enough to do a good design, and secondly, even if you could, the user will change his mind, or the very existence of your system will suggest new ways of doing the job better. Rule 3 suggests an important design strategy, **iterative design**, which is the *deliberate* attempt to design, test, and design again.

User interface design is only visible when a system can be used, and the standard ways of evaluating interactive systems do not generate useful data for design, as opposed to *re*design.

One can measure the user's training time, his error rate, his productivity and so on, but if these are unacceptable it is not necessarily clear what part or parts of the design should be reappraised. And, besides, once a system has developed far enough to be evaluated, too much software investment has gone into it to make the sorts of radical changes that may be needed.

Given this dilemma, the two best ways to design a system are:

- To copy or develop an existing system known to be successful. This approach has several merits, not least that of reducing user training.
- To design the system abstractly, making full use of prototyping tools, to minimise the investment in development before it can be evaluated.

Conventional user interface design relies heavily on evaluation by users: put cynically, the design is not perfect, but the users will be able to fix it. This tends to result in bottom-up design, as users provide information about isolated aspects of the system: this needs changing, that needs adding. If a design is to survive the insights of evaluation—useful as they may be—it must have a structure by which suggested modifications can be judged. Thus some modifications will be inappropriate, but others may have consequences elsewhere in the system that the users could not have reasonably anticipated. Top-down, principle-led, design is much harder than bottom-up *ad lib* design, and it is only comparatively recently that it has become a plausible way to design interactive systems of any complexity. This advance is primarily because of the leverage provided by formal methods at the early stages of top-down design. At present, much of the application of formal methods in interactive systems design is still at the research stage and probably cannot be used very effectively for large projects. Nevertheless the 'motto' of formal methods—thinking attention to detail—can most profitably be adopted in user interface design. Formal methods is not merely pedantic description, but by expressing ideas lucidly gives an opportunity to test those ideas on paper and, if they fail, to refine them. Without formal methods, or at least a formal inclination, you get none of: the clear expression of ideas, the attempt to prove ideas, the attempt to refute them, the attempt to improve on them [6]—and hence none of the improvements.

3 Choose a style

Different applications and different available hardware (and different levels of manpower to implement the system) impose certain constraints on the interface. There are three broad catgories of system:

1. *Special purpose*—
   (a) *industrial*—e.g., aircraft or military vehicle cockpit control systems. Here special displays (head up displays, dedicated instruments etc.) and input devices (joysticks, knobs etc.) are developed. An important consideration is that the user is often involved in multiple tasks possibly under extreme pressure and will not be able to handle overly-complex systems. Generally, the relevant industry will impose strict standards.
   (b) *individual*—e.g., to help people with special needs. Although special hardware is readily available (micro switches, breath control devices etc.), typically the hardware and software must be tailored to the individual. An important consideration here is that the user will often be exhausted after 'small' efforts to use the system. Paradoxically, predictive systems (that try to anticipate the user's next action, to save him the effort of doing it) can be counterproductive.

2. *Simplified*—e.g., public operated devices, such as cash-dispensers, travel enquiry booths, arcade games and complex consumer products generally, such as video recorders, music synthesisers, calculators and so on. An important consideration with simplified systems is that users are generally **discretionary**, that is, they choose to use the system (and probably pay for it), and will only continue to use the system if it does its job well.

3. *Workstation*—any application requiring a general purpose computer, typically relying on text and hence on typewriter keyboards. Workstations now have *as standard* screen resolutions sufficient for graphic effects, sound output sufficient for acceptable speech, and analogue data entry devices (e.g., a mouse or touch screen) to supplement QWERTY keyboards. An important consideration with workstation systems is the 'obvious' fact that workstations can do practically anything, and therefore they *may* be doing anything. The user therefore requires additional cues as to what is happening—cues that may not have been necessary for special purpose designs with characteristic controls.

Workstations provide the greatest range of possibilities, indeed they may be used to prototype or simulate interfaces from any other category without the expense of tooling up for special hardware. Conversely, workstations may inhibit better, but more creative, design! Few musicians, for instance, would be satisfied with a workstation style interface over a piano-style keyboard.

Within any category of system, there are four broad **styles** of interaction.

1. *The system provides choices for the user*. Here the user has little to remember, except how to choose from among the alternatives presented. This style is therefore ideal for **infrequent** or **casual** use. Example: **menu systems**, where the system provides a list of alternatives (e.g., destinations for a sight-seeing trip). The user can select from the alternatives in various ways: they may be numbered; they may have buttons adjacent to them; they may be displayed on a touch-sensitive screen. Note that the menu need not be textual, but can be composed of pictograms or even sounds. In the special case that the user is allowed to relocate the menu components, giving a certain sense of physical concreteness, the style is termed **direct manipulation**.

2. *The system provides a structure for the user*. Here the user is allowed more freedom than merely selecting from alternatives, but nonetheless is 'led through' the interaction, with the computer in control. The user again has little to remember. Example: **form filling systems**, where the system displays boxes, each of which requires certain data.

   It is very useful to distinguish between **spatial** and **temporal** structure. A temporal structure usually manifests itself as a series of questions and answers—the computer asking the questions. The user is very restricted, particularly if he notices an earlier mistake that has 'misled' the direction of questioning. Care must be taken to permit the user to go back over earlier answers, unless this style of interaction is employed for security reasons (e.g., what is your name? what is your clearance? what is your password?). In contrast, a spatial structure allows the user to provide answers in any order, and they can see all relevant questions (and their answers!) at once. (Note that a series of spatial structures can be imposed on the user as a temporal structure.)

   The computer can often use the context (e.g., prior knowledge of the user's task, or of his previous answers) to suggest the user's answers. Such suggestions are termed **defaults**. In a temporal structure, the user will require an 'accept default' button, but in a spatial structure, no special provision for defaults is necessary—the user can simply avoid changing the default.

   Figure 1 shows a typical temporal structure in the **question-and-answer** style. The four questions are asked in order, and no question is asked till the previous one has been answered. The question, `copies?` must be asked first, since if the user answers `0`, the other questions need not be asked. Note the arbitrary abbreviation `i` for inches, and that each answer must be followed by a **delimiter**, in this case shown as $\hookleftarrow$, but often the ASCII character return or escape.

   Note that once the user has typed `i` $\hookleftarrow$, he is not given an opportunity to change to mm. It might have been better to ask the user to type each measurement with its own units, but this would have been not only more tedious but would have relied on the user remembering the more complex form of answers required.

   Contrast this with Figure 2 which shows the equivalent spatial structure, in this case a so-called **dialogue box**. The number of copies to be made has not yet been specified; the user can request

```
copies?  1 ↩
inches or mm?  i ↩
width?  14 ↩
height?  7.75 ↩
```

Figure 1: Temporal structure: question-and-answer style

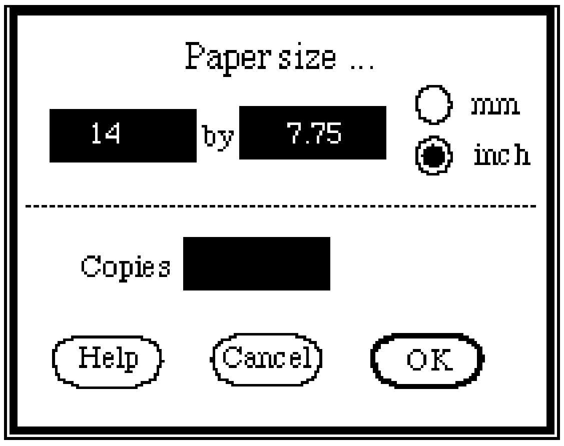

Figure 2: Spatial structure: form filling style

help, can cancel this paper specification, or can request the computer to proceed (by pressing '`OK`'). Note that the **fields** can be filled in in any order, for example, the user can change the units of measurement before, after or even during entering the numbers. The dialogue box is probably best used with a keyboard (for entering the numbers) in conjunction with a **pointing device** such as a mouse, however this is not necessary. For example, pressing `tab`, a certain key, could move the typing position between the various **fields**, or alternatives. Since all data the user enters is expected to be numeric, typing `H`, `C` or `O` can be used to select help, cancel, or OK as appropriate. There may, of course, be other conventions for help, cancel and OK and these should be used—note that abbreviations such as `C` for cancel are language specific, and may not work if the system is exported to another country.

3. *The user provides 'free form' input.* Here the user is assumed to be skilled in the application. Examples: a drawing system allows the user to draw lines and curves freely on the screen; a **command-based** system allows the user to type statements in some language. In both cases, the computer will impose some syntactical structure (e.g., about how lines may join into polygons), but in distinction to style 2, the parsing happens *after* the user has submitted input. Since there are so many approaches to parsing, little specific can be said about style 3 in this chapter, except that the designer should take advantage of known parsing algorithms (including language processing tools such as compiler-compilers) where ever possible.

   The previous example of paper-size setting is accomplished by the user typing, say, `papersize 14 7.75` (assuming inches are defaulted). Note that the command can in principle be entered at any point where a command is expected: the user does not have to wait for questions or a dialog box to type `papersize`. On the other hand, the user has to remember the command name for setting paper size (here, `papersize`), and the correct way to use it. Usually one command, say `?` will print out all available command names, and composite commands like `?pap` will print out information on all commands starting with `pap`, or containing `pap` in their name.

   An alternative way to specify paper size is to use **gestural** input, rather than by using **symbolism** which has so far been illustrated. With a gestural interface, the user will provide input by some analogue means, for instance by drawing a full scale rectangle the size of the required paper (e.g., by using some suitable measuring device, like a tablet)—thus avoiding the question of units of measurement, but however introducing potential inaccuracy (or even permitting the user to specify non-rectangular paper!)

4. *The user can program the system.* A rough distinction is usually made between programming **in** the system (increasing its functionality) and programming **on** the system (merely providing aliases and short-cuts, **accelerators**). Spreadsheets are a good example of programming in; programming on is usually provided by a **macro processing** scheme (and is often also available in spreadsheets) or more simply by **customisation**.

   Macros allow the user to define certain symbols (keystrokes or words) to expand to useful sequences, primarily to save typing effort, indeed, to save memorising long sequences. The advantage of programming on is that it can be implemented separately, as-it-were lying between the user's keyboard and the underlying application; the disadvantage of programming on is that it can interfere with the underlying application, for example by ignoring any syntactical constraints that should be imposed. In particular, consideration must be given to the manner of handling error messages that arise during the use of a macro—that is, not directly arising from the user's explicit input to the system.

   Again, the designer should always take advantage of accepted styles of programming language, and beware of the standard traps (e.g., the dangling else problem). Far too many systems have been marred by *ad hoc* and incomplete languages when a subset of Pascal, Forth or LISP would have been quite adequate.

Since the four styles of interaction represent increasing flexibility for the user at the expense of greater load on the user's memory, it is sensible to provide **help**, a means to simplify the style of interaction temporarily. For example, in a style 3 interaction, the user may 'be lost for words.' Pressing a help button (preferably one permanently reserved and engraved as such on the keyboard, or 'virtually' displayed on the screen) could bring up a menu of choices, or if this is not possible, of documentation that the user can browse to understand his problem.

4 Use principles

It is generally held that to use principles in user interface design is better than not using principles, even if these principles are arbitrary. Systems are easier to use if their component parts are mutually consistent even when their parts are unconventional—for example, the complete system should only use one algorithm for abbreviating long names, whatever that algorithm. However, there are plenty of approved principles to deploy (including algorithms for abbreviations) and there is rarely any need to diverge from conventional principles.

The reader is referred to [11] for a substantial collection of principles. Here we give a very brief list, omitting such 'obvious' principles as providing rapid feedback to the user's actions. Notice how principles are not necessarily consistent with each other in the limit: the designer has to seek resolutions dependent on the particular circumstances. Also, in certain circumstances there are good reasons to flaunt these principles: for example if security is an issue, it is better if the user cannot use the system at all than be reminded of his password! Intriguingly, systems that look identical may need quite different design principles: arcade missile games and *real* missile control systems being a case in point.[1]

4.1 Be consistent

Everything should work the same way. This is surprisingly difficult to achieve, not least because it limits the ways in which an existing system can be improved by specific enhancements. Consistent user interfaces avoid the severe problem of **carry-over**, that the user's skills learnt on one system are 'carried over' onto the present system, whether appropriately or not. Carry-over happens particularly when the user is under stress, say, after causing a serious error, and may in turn cause further errors.

Consistency applies to output: the symbols used and screen layouts should be consistent. A good visual consistency is often called a **style** and can be used in a variety of situations; a style may be explicitly designed, e.g., by a typographer.

Consistency also applies to the relation between input and output, which is the next principle:

[1]In this comparison, a careful distinction must be made between *excitement* on the one hand and *alertness* on the other.

4.2 Enforce compatibility

Input and output should be compatible, so that in principle the computer's output could be supplied as its own input and *vice versa*. For example, if the system asks for a date in a certain format (like `10 January 1989`) then it should also display dates in this same format throughout.

4.3 Use confirmation

1. *redundant coding*—Contrary to the principle of compatibility, when the user provides input, it is useful if the computer confirms the data *in*compatibly. For example, suppose the preferred date form is `10/1/89`, if the user submits `3/7/89` it is useful for the system to confirm the date as `3 July 89`, in case the user thought he had entered the date March 7, 89.

2. *checkpoint*—ask the user whether he wishes to proceed with costly or irreversible actions. Figure 2 shows checkpoints 'cancel' and 'OK' in use. Be aware that the user's response may become automatic if he too often needs to reply 'OK'—some other style of confirmation may be appropriate, or (better still) make more actions reversible so that they do not need confirmation.

4.4 Be clear

Be *very* careful about wording. For example, "`Quit without saving data:  yes, no or cancel?`" is ambiguously worded. Putting the question positively ("`Save data before quitting?`"), although good practice to improve clarity does not help in this case: the user might say `no`, thinking he did not want to quit, but even so the computer might quit, 'thinking' the user did not want to save! (In this case, it would have been better if confirmation was not a simple answer, but a command, as in "`If you really want to quit without saving data, type REALLYQUIT`".)

It is advantageous if all phrases can be configured in case they later turn out to be inappropriate, e.g., by having a file of standard messages. This simple technique, incidentally, not only makes it easier to export systems to foreign countries but also allows users to work with specialised terminology if necessary (consider a stock control program used in different sorts of business).

4.5 Minimise data entry

The less the user has to enter, the less likely the user is to make mistakes. Nevertheless, errors will still occur, and minimising data entry should not be taken so far that accidental input has devastating consequences.

4.6 Provide for flexibility

It is hard to anticipate in what order the user will want to accomplish a task, in fact, the user may perform the same task in different ways on different occasions. Systems should therefore not have a fixed order for progressing through a session. Word processors are the classic example of flexibility: a word processor allows the user to write his document in any order: starting from the beginning working to the end, starting from headings and fleshing sections out, or even adding bits here and there in any order.

4.7 Reduce display clutter

Put as little as possible on the screen. Do not fill the screen with lots of unrelated information (and don't use lots of colours—even allowing for colour blindness).

4.8 Make inertial displays

Change the screen as little as possible (the principle of **display inertia**), and keep regions of it effectively constant (for example, titles). In particular, if menu selection causes side-effects (such as highlighting) then when a menu is redisplayed it should show its last highlighting. This often gives the user a helpful indication of what he has most recently done in that menu.

4.9 Exploit redundant information

For example, order entries in a menu alphabetically, then the user will be able to locate them faster. Or allow users to select from menus by alternative means, by typing the entries (abbreviated or in full), by using their key numbers, or by pointing at them.

4.10 Allow for closure

Closure is best illustrated by example: when a user goes to a cash dispenser (autoteller), they will complete their task—reach closure—when they get their money. The user first has to push their plastic card into the machine, type a few numbers, then get their money. At this point the user will have reached closure, and may walk away *leaving his card behind.* It is therefore essential that the user is given his card back *before* he has reached closure; that is, the user should be given his money as the last step after all other details have been completed.

4.11 Provide context sensitive help

If you want to know what I mean, type `help` ...

There is a great deal of advice available about writing good and effective help (see the bibliography). Briefly, make help affirmative or positive, "*do* this, *do* that" rather than negative, "*avoid* this, *never* do that." It also helps greatly to order actions even within sentences in exactly the order they are required; for example, don't write, "eject the disc after checking you have saved your data," but write "check you have saved your data, then eject the disc."

Note that when writing manuals that your perspective is different from the user's. Typically, the user comes to a manual with a problem and wants to find a solution: that means that indexes, cross-referencing and so on, should tell the user what things *do* as well as what they are called. Who, for instance, would think of looking up `rm` if they wanted to delete a file?

Users often work in groups, and you may want to provide a specific mechanism so that they can help solve each other's problems, for instance by a simple form of email or 'complaints' field.

But the most useful advice is simply: make help as brief as possible (this will, of course, be easier than writing copious material!) ... then simplify the system so that what you have written is sufficient!

4.12 Instrument

Provide means to record what the user is doing, how long it takes, what errors are being made, and so on (with due regard for privacy of personal data). Instrumentation is very useful in identifying bottlenecks in system usage, and may even be retained once a system has been fully released. For example, if a system crashes or has other unusual problems, the logs may provide the designer with useful clues.

4.13 Provide undo

Users make mistakes and will frequently want to undo their actions. The computer, too, may have bugs that cause undesirable effects. In both cases, it is helpful to be able to undo the steps that cause the problem. In the case of computer bugs, the only general way is to keep a **log** of what the users does: when the system crashes, the log can be replayed just up to the point of disaster. This may then enable the user to recover his work by some other method, if the crash was indeed caused by his actions, rather, than, say a power failure.

Note that many systems provide a cosmetic sort-of undo that has no real power to recover from errors. For example: a drawing program might have a rotation command; although any rotation can be undone by the undo command, the user can 'undo' any rotation merely by rotating backwards, yet he may not be able to recover accidental deletions. Note that undo should be able to delete more than just the most recent action: it is probable that after deleting everything the user will panic, and not immediately try the undo command.

4.14 Be non-preemptive

It is not uncommon for dialog boxes to ask a question while obscuring the information the user needs to answer the question. A preemptive system would require the user to answer the question *now*, perhaps even to the extent of not allowing the user to move the dialog box to reveal the necessary information. A word processor may have a 'check spelling' feature, and as it is running—helping the user locate spelling mistakes—it should not preempt the user, so stopping him making any other changes he now sees to be appropriate as he fixes spellings. Here preemption is due to the computer only allowing one task (spelling checking), but the user trying to do two (spelling and editing). Preemption may also occur when the user has explicitly embarked on several tasks, say, drawing a picture and printing the previous picture. If the printer runs out of paper, the system can warn the user, but need not preempt the current drawing to tell him.

4.15 Allow interruption

Conversely from §4.14, if it is a good principle that the system never preempts the user, it is a good principle that the user be able to preempt the system. The system may embark on some long operation, but the user now wants to do something else. The user should be able to interrupt the system at any time, and to resume the interrupted activity when appropriate.

Note that good **windowing systems** simplify the implementation of non-preemption and interruption principles.

4.16 Be modeless (or low-mode!)

Each mode is a different way of interpreting the same action. The keystroke `R` may mean run, replace, repeat, rotate, or it may be just the letter `R`—and then it may mean *insert* the letter`R` or *overwrite* with the letter `R`. The more meanings any action has the more likely the user is to misinterpret it. Modes of course allow many actions to be made available from a limited symbol set (e.g., the QWERTY keyboard), but great care should be taken that a frequently used symbol in one mode is not over-loaded as a dangerous symbol in another mode. It is generally best if non-standard modes are brief, e.g., only active for a short period (e.g., for one keystroke).

Note that different windows strictly represent different modes; the spatial separation of windows is a great aid reducing mode errors, but it is still a frequent error to 'type into one window while looking at another,' that is, to enter data in the wrong mode.

4.17 WYSIWYG (what you see is what you get)

Many systems, particularly desk top publishing and direct manipulation systems display a picture representing something real, or potentially printable. At its simplest, WYSIWYG requires that the screen display is an accurate rendition of what will be printed. This is harder than it seems, for instance because of differing resolutions (say, 100 dots per inch on a screen against 300 on a printer), because of different colour schemes (additive on the screen, subtractive on a printer), because of slight differences in font metrics—people being very sensitive to typographical issues.[2] Note that the user may make implicit judgements, say about the alignment of text in certain fonts on the screen, which he expects to work just as well on the printer. More strictly, however, WYSIWYG requires that what the user sees on the screen is *just* what he has, no more and no less. In a word processor, for instance, this would have consequences for the handling of blanks (tabs, newlines and spaces) since they cannot be told apart, all being displayed as nothing. See Figure 3, which shows grey regions that the user has *got* but cannot *see*—a common design error that can easily be fixed.

A direct manipulation system adhering to WYSIWYG would not use accelerators, i.e., keyboard equivalents of certain actions with the mouse on visible objects. The accelerators are not visible (in fact, the entire keyboard is visible, so no *particular* key is visible!)—an advantage of direct manipulation, visibility of possible operations, is lost with keyboard accelerators.

Simple hand-held calculators provide another example of failed WYSIWYG. The display shows `84`, say. But what you see does not tell you what you've got: the calculator may have 'got,' `84` itself or `84+` or `84-` or `84=` or . . . , and the effect of the next keypress is not determined.

[2]E.g., Figure 3 had to be edited and printed four times to align the gray regions, even though the alignment appeared correct as displayed on screen!

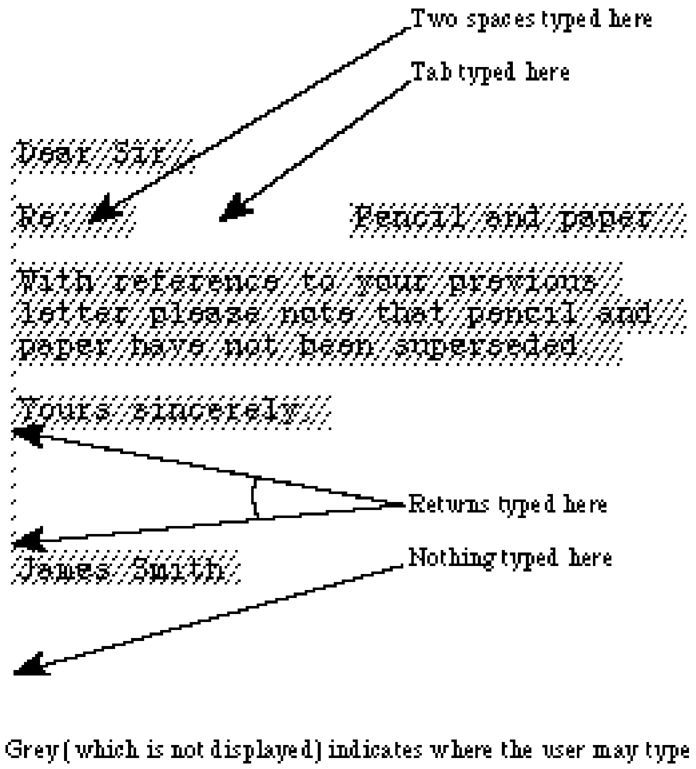
Dear Sir
Re Pencil and paper
With reference to your previous letter please note that pencil and paper have not been superseded
Yours sincerely
James Smith

Figure 3: Incompatibilities with WYSIWYG

4.18 Minimise user's memory load

Don't assume that the user can remember how to use the system ... the user may anyway be an infrequent user of the system. Always provide some form of help, adopting whatever standard is used in the rest of the user's environment (e.g., hitting the `?` key).

The calculator example above (§4.17) shows that non-WYSIWYG displays rely heavily on the user's memory to know what keystrokes have gone before.

Use **mnemonics**—names—for things in preference to numbers, and chose a consistent way of naming things (including whether capitalisation is significant). There is no excuse to make the user remember 'magic numbers' or numeric IDs. (A macro processor can help sanitise simple interfaces by permitting names to be defined as numbers or other un-mnemonic symbols.)

4.19 Provide a sense of progress

Except for skilled users (who were anyway once unskilled), users require 'encouragement' that they are getting closer to achieving their goals. Show how many pages remain to be printed. Show how many words have been typed. Show how far the spelling checker has got through the report. Show how long rendering a car body has yet to take. In some cases absolute measures will be required (e.g., word counts, minutes) in others a percentage is sufficient. It is, of course, possible to provide both! Figure 4 shows a simple per cent completion bar, shown both graphically and digitally.

Such indicators also help the user when (unfortunately!) system limits are reached. Without suitable measures, the user can only guess how to reduce memory loads, but with a memory-remaining indicator, the user has a good idea how effective his 'compaction' is, and also at what stage it is not worth trying to enter more work.

4.20 Commensurate effort

Every user action causes a **response** from the computer, some of which is visible, some of which is (for the moment) internal. The commensurate effort requires that the user's effort in getting the computer

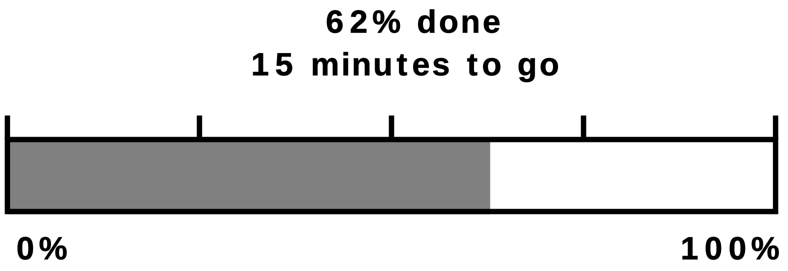

Figure 4: Per cent completion indicator

to do something should be commensurate (roughly, in proportion) to the work the computer does. In particular, it should not be possible to *destroy* information appreciably faster than the user can enter it.

Here is an example that shows using principles creatively improves interfaces. A desk top publishing program permits the user to change the point size of his text. Typically this can be done on letters, words or larger sections. Consider changing the type size of a large section, e.g., the entire document. The commands available may permit the user to set the point size to 10, 11, 12 etc. But, if so, then other changes to point size (e.g., in subscripts) within the document will be lost. This is not commensurate effort—information created by many commands (for each of the previous size changes) is lost. Instead, the design should cater for the user to increase or decrease the point size (say by ± 1 point): these actions, incidentally, are reversible.

4.21 Ask, "Can a computer use it?"

Designing systems for fickle people to use is hard! They change their minds, and probably never really know what they want until after the designer has implemented something else. A useful and rigorous test of an interactive system is to try it not on people, but instead on a computer (theoretically or actually). Thus, the user interface, in order to be usable by a computer, must not rely on 'insight' or unrevealed knowledge. Any questions asked the user must be computable—if the answer is in principle known by the system (such as the date, or a part number), then it should be accessible for the computer 'user' by some well-defined mechanism.[3] (If we follow a non-procedural programming paradigm, we might further want every 'user' response to be in principle computable from the task what is visible on the *current* screen alone, rather than in addition from information held in the user's memory of past screens.)

A word processor, for instance, must not conceal information that it 'expects' its human user to know (e.g., whether a certain region of blank screen is spaces or 'really' empty), for a computer 'user' certainly will not. Of course, a computer *could* know enough to use any system merely by duplicating that system's program: but then that would be like expecting the user to know the system's program by heart, an unlikely situation (except for the designer). If the system has reached some limit (e.g., the user has typed 6 characters into a 6 character code field), then it should indicate that the limit has been reached. How is the user supposed to know? Would you want to program in all such little details into the test computer program?

The 'Can a computer use it? Test' helps remind the designer that his task is not just to design an interactive computer system, but he must also 'program' the user—provide good enough documentation—so that the system can be used correctly under all circumstances.

4.22 Be wary of natural language input

Natural language has the impression of being easy to use, and in some sense it is. Unfortunately parsing natural language is extremely costly, slow, and except in heavily circumscribed applications subject to ambiguities. Of course the system can always ask the user to clarify which of several meanings he had in mind (a form of confirmation), but too often the apparently unlimited scope of natural language will lead the user to impute impossible skills to the computer. Nevertheless, one should not underrate trivial language processing, as in many adventure games (as in, '`pick up the red ball`'), or basically

[3]Thus questions must not be preemptive—for if so, then there is *no* mechanism.

‘deceptive’ language processing, where the computer merely looks for keywords, simply ignoring words not in its vocabulary.

4.23 Think of wider issues

Concessions will generally have to be made for international markets (e.g., for date formats and for the language used for any natural language text) where appropriate. Thus standard chess notation, representing Knights by `Kt` or `N` (which should be made **aliases**—alternative names for the same thing—in an interactive system), is not the accepted international notation: so-called figurine notation is preferred, where Knights become ♘ and are then unmistakable in any language—and can also be recognised by people unable to read. So, make full use of *clear* symbols (**icons**) where possible—though note that the editing and layout of symbols is not so easy as text (e.g., there is no alphabetic order).

A surprising number of users are handicapped in one way or another (many of us cannot add up numbers accurately; many men are colour blind; many users cannot type very well; many users are doing more than just using the computer—dealing with clients) and providing suitable features enhances the system generally.

Initially every user of a system is a new user, so provide features to support new users. A recommended way to do this is to provide **minimal systems**—systems that are syntactically complete but semantically restricted. Thus menus will show the user what he can do on a full system, but only safe features are available on the minimal system. The user can use a minimal system quite safely, but is also naturally exposed to the range of features of the full system. Menu entries can be ‘greyed-out’ if they are inactive, but in general it is better to let the user start advanced features with a warning that they are disabled but would have done such-and-such (or perhaps, ask for confirmation: the user may want to learn by exercising on a disabled feature, or may really want to use an ‘advanced’ feature though still a new user).

4.24 Allow for catastrophe!

The computer will someday crash, there may be a power failure, a virus or some hardware failure. Therefore build in mechanisms so that the user can recover ‘lost’ work: keep logs of the user’s activities so that they can be replayed on another working system; don’t allow the user to work for a long time without making backups of the current session. *Following the other principles in this list will undoubtedly bring ‘percentage’ improvements to any interface; following the final principle may save the user weeks’ worth of work—perhaps his company as well.*

5 Gueps: dual principles

Many of the design principles listed above have an uneasy status. It is not *quite* clear what they mean for any particular design, and yet they are undoubtedly ‘true.’ Some principles relate to the complexities of the real world (e.g., design for international markets), but most concern the inter-relatedness of components of the user interface. As such, they may be formalised and their application in a particular design reasoned through.

In any contractual situation or where a design team of several people is involved, it is essential to identify what such principles really mean. Generative user-engineering principles, **gueps**, are dual principles that have both formal and informal forms. A principle such as WYSIWYG can be expressed as a theorem of the system specification, and can also be expressed colloquially in a form suitable for a user to understand—WYSIWYG itself is practically a colloquial term now! As a theorem it can be shown to hold (or not) as the case may be, or to interact or conflict (or not) with other formalised principles. The advantages that accrue are precisely those of formal methods: the designer knows exactly what is being designed and what its properties are, and furthermore those properties are (presumably) relevant to the user. In particular, the informal expression of the principles can be adopted in the manuals or other user-training material (including on-line help) as ‘golden rules’ about how the system behaves.

Figure 5 indicates the idea: the design principles are first agreed and then expressed as dual principles, both for the software engineering of the computer system design, and for the ‘psychological engineering’ of all the other materials (training courses, manuals, etc.) that are part of the product. The software engineering and pyschological engineering proceed more-or-less in step (though obviously some revisions may be forced from either side). The end results are respectively a computer program,

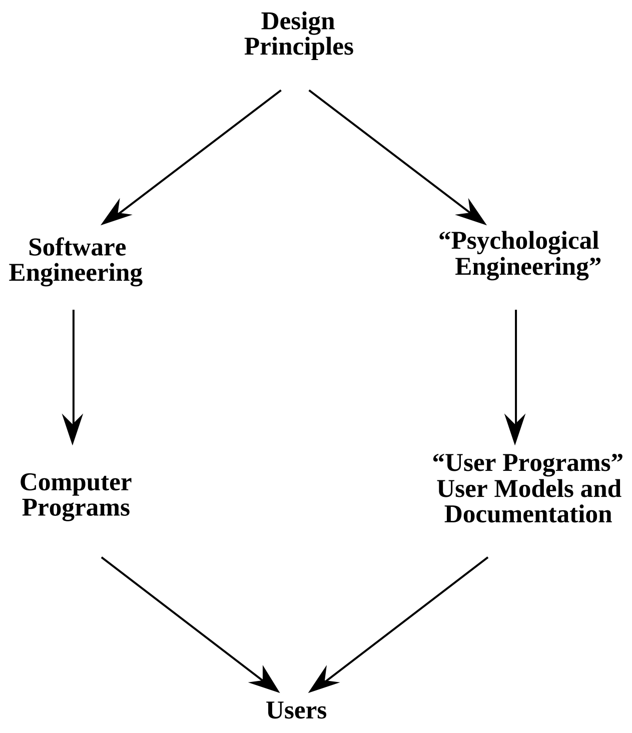
Design Principles
Software Engineering
"Psychological Engineering"
Computer Programs
"User Programs" User Models and Documentation
Users

Figure 5: Top down design with dual principles

adhering to the principles and a 'user program'—a manual, for instance—adhering to the *same* principles. The system and its manual therefore agree unusually closely, especially in the nature of limitations, boundary conditions and 'bugs.' In contrast, a conventional design method would have the manual being written by professional documentation writers *after* the system was complete, probably at the last possible moment (to avoid the cost of revising the manual as the system undergoes its own revisions). This means that insights gained in the 'psychological engineering' of composing good documentation come too late and have to be ignored.

More about gueps can be found in [13]. An application of them is given below in §7.

5.1 Exploit programming language principles

The first problem with gueps is thinking of them! There are two approaches: invent your own, which is hard, or steal some. A fertile area for stealing them from is programming language design, which we now consider.

Programming languages such as Pascal have come under constructive criticism from denotational semantics. The mathematical way of looking at programming languages means that some things are equally easy to say mathematically, but the language for one reason or another has variations or limitations. Thus, Pascal can only make functions out of commands (so-called `procedures`) and out of expressions (its so-called `functions`). But Pascal involves other constructs, such as declarations, and once it has been 'mathematicised,' there is no reason in principle not to have functions of declarations—**classes**, in other words. This is the **Principle of Abstraction**: that meaningful syntactic categories (expressions, commands, etc.) can be abstracted, made into procedures. Related programming language design principles are the **Principle of Correspondence**, the **Principle of Qualification**, the **Principle of Orthogonality** and the **Principle of Data-Type Completeness**; see [12] for examples. Such principles can be used in user interface design, particularly when the user interface provides a rich variety of features.

The Principle of Correspondence is perhaps the easiest of these principles to apply to interactive systems design; however its application is first illustrated by an example from Pascal. In Pascal there is a semantic correspondence between variable declarations and formal parameter declarations. Thus the two pieces of Pascal code in Table 1 correspond exactly: both fragments bind the name `x` to new storage, initialise it to 3, and invoke `write` (representing the arbitrary body of the block).

`var x: integer;`	`procedure p(x: integer);`
`begin`	`begin`
`        x := 3;`	`        write(x)`
`        write(x)`	`end;`
`end`	`begin`
	`        p(3)`
	`end`

Table 1: Pascal declaration correspondence

miniFinder	*Finder*
new file	—
open file	open file
open directory	open directory
view directory as text	view directory as text
scroll directory	scroll directory
—	view directory as icons
view directory alphabetically	view directory alphabetically
—	view directory by date
go to any parent directory	—
—	rename existing file
—	change directory of existing file
—	copy file
eject disc	eject disc
—	rename disc
—	use menu bar, run other applications etc.
—	get information on files
find file within directory from first character[4]	—
—	find file anywhere from full name

Table 2: miniFinder/Finder correspondences

In general, the correspondence for any such declarations is exact, assuming only that `p`, the arbitrary name of the procedure, is not otherwise bound. The correspondence could be 'tidied up,' by permitting variables to be initialised when declared (e.g., `var x := 3 integer`. Furthermore, the correspondence has not exhausted all declaration and formal parameter mechanisms, and in Pascal there are mechanisms that do not correspond. Thus, `var` parameters have no corresponding declarative mechanism (though it has semantics similar to `with` declarations), and `const` definitions have no corresponding parameter mechanism. Of course, the revised ISO Pascal standard has addressed some of these issues, for instance, providing constant parameter forms (but has not permitted constant definitions with expressions, which is immediately suggested by the correspondence of being able to pass constant expressions as actual parameters).

Now consider a user interface such as exhibited by the Macintosh Finder and miniFinder. The Finder is the 'operating system level' of the Macintosh, allowing users to find and open files; the miniFinder is an alternative mechanism that allows users to find and open files when they are running an application. Clearly these two schemes correspond—the correspondence is, however, slight, and could easily be completed ('closed'). Table 2 shows the extent of the correspondence, and *some* of the missing possibilities.

Overall, then, adopting programming language principles not only inherits a wealth of powerful research into expressiveness and consistency, but encourages a more systematic interface design.

[4] Only when opening an existing file, not when creating or saving a file, when typing changes that file's name.

6 Use tools

So far, the discussion has been conceptual. We now briefly review practical programming and evaluation tools.

6.1 Programming tools

A user interface, say, for a database requires: basic input/output (graphics, mouse and keyboard event handling), 'design' (what should the screens look like?), semantics (what should it do?), data (what does it do it with?) and integration (how does it work with other systems). Often a 'minor' change in the appearance at the user interface level will require simultaneous changes at all these levels. Obviously this means that you should try to avoid changes ('get it right first time,' perhaps by adhering closely to standards), *or* use a programming environment where there is little overhead moving between the levels.

Object oriented programming, particularly in Smalltalk which includes powerful graphics primitives and standard procedures for interaction techniques (such as menus), is an ideal approach. Unless the development is one-off, the disadvantage of Smalltalk is that it requires considerable computing resources which will almost certainly mean that the application has to be recoded in a conventional programming environment once it has been finalised. But this is a very minor cost, given the flexibility available and the ease of getting the first implementation to work well-enough for trials. Object oriented programming for user interfaces has the very significant advantage (for producing quality interfaces) that most user interface features correspond to specific fragments of program—**objects**, in fact. Thus improving the program tends to improve the interface, a relation that does not obtain in other programming paradigms.

HyperCard is a simple programmable database with simple bit-mapped graphics and is ideal for developing experimental user interfaces. The major contribution of HyperCard to user interface design is the ease of moving between the various levels of implementation. Furthermore, HyperCard is extensible and can be augmented—programmed *in*—with 'external commands' written in conventional programming languages such as Pascal. Unlike Smalltalk, HyperCard runs well on cheap hardware (Apple Macintoshes) and has inspired a number of imitations on other computers. For some applications a HyperCard system will be quite adequate as a final system, though error handling is often tricky, if not actually impossible in some cases. Note that in general the chosen programming system's exception handling mechanisms can have a significant effect on the style and quality of the final interface.

There are a number of interface design tools, called UIMS (User Interface Management Systems). These tend to impose a specific style of interaction on applications: they are not equally suitable for all jobs. It will help if the UIMS generates source code (e.g., in Pascal), in case subtle changes need to be made by hand, though this step may compromise maintenance at a later date.

The distinction between UIMS and normal programming support environments is becoming eroded as programming environments become more sophisticated. Thus systems such as interface libraries, MacApp, X/windows, even spreadsheets and database systems, provide many features overlapping with UIMS. Some text editors (e.g., EMACS) are programmable and can be extended almost indefinitely, especially useful to prototype text processing applications. A good UIMS, however, should not only provide an environment for rapid prototyping, it should also provide some *analysis* of the user interface (can the user get trapped? does every sub-system have the same exit command? how many commands are needed to get from here to there?) and be able to generate code that can be incorporated into the rest of the application. (Or *vice versa*: maybe the application can be incorporated into the UIMS code.)

At all levels of design it should go without saying that appropriate programming tools be employed: parser generators, lexical analyers, etc. Do not underestimate the power of those tools, pencil and paper—both for thinking and for prototyping. Users can often help assess a very prototype design shown them just using hand-drawn illustrations and hypothetical questions, "You did that, now the computer will show something like this, and ask you to do this."

6.2 Assess

Most users know more about what they are doing than designers (though designers may have useful insights). Only users know why *they* are puzzled, only users know what they wanted to do, only users know what they *thought* was happening.

A very simple technique, the **think-aloud method**, is a powerful and cheap way of evaluating user interfaces. Put simply, the think-aloud method is to get a designer to sit down with a user who is working with the system in question and to encourage the user to think aloud about what he is doing. The user is encouraged to say whatever comes into his head as he tries to use the system.

The advantage of sitting with the user and encouraging him to think aloud as he works is that many things would otherwise be quickly forgotten. Giving the user a questionnaire (a standard technique, but requiring careful design and administration) afterwards may be too late. In fact a new user may not know whether he is making mistakes, but the watchful designer can make a note of any problems.

A few simple rules for the think-aloud method need to be observed:

- The user must be encouraged to feel at ease thinking aloud. The designer should make it quite clear that this is a test of the system, and not a test of the user! Thus every problem the user has is valuable to hear about.
- The think-aloud method creates an artificial situation: the designer has to be there, but he must not give special help to the user even if he is asked for it. This may seem rude, and the designer should explain and apologise in advance. The designer should explain that he needs to see how the user sorts out problems for himself. Nevertheless there will be times when the designer needs to intervene and provide hints: this will be when the difficulty has been noted, and it is apparent that the next design can fix the problem—there is no point holding the user up any further.
- It is useful to remember that being helpful is a temptation to be avoided—especially because you as designer know far more about the system! The more help that is given, the less useful the think-aloud work will be about the *current* design being tested.

Think-aloud can be used not only for helping debug the computer system, but can also help with the design of documentation, and to help discover discrepancies between the documentation structure and the way people use the system to perform their tasks.

Think-aloud is so effective in leading to design improvements that an n person design team will often better split up as an $n-1$ member team, with the 'lost' person doing something else. When the system is almost ready, the 'lost' designer returns to be the 'user' for think-aloud debugging. Because he has not seen the system, *and has not been party to the reasons—or excuses—why the system has turned out the way it has*, he will have startlingly good insights into the user interface design! The end result will be far better, and can be done with fewer man-month's work. (Clearly a designer will tend to generate different sorts of insights about the system than a more typical user. They will still be interesting but may be more technically oriented.)

6.3 The Wizard of Oz

Instead of implementing a complete system, for some purposes—particularly during development—it is sufficient to *pretend* to implement a full system. A human, called the wizard, monitors the user's interaction and intevenes as necessary so that the user sees a more-or-less fully developed system. Figure 6 shows this diagramatically. A simple application of the wizard is to convert prototype error messages (such as `Error!`) into something more relevant to the user's task (such as `You can only print one master copy`). Presumably, the human wizard knows that the user is trying to print a master copy, but the prototype system does not yet recognise the context. When the wizard's interventions are examined, various ways of enhancing the interface become apparent. A very powerful way to use the Wizard of Oz technique is to simply place *untrained* users in front of a system and tell them to use it, to print invoices or whatever. Here the users have no preconceptions about the syntax or vocabulary, and the wizard must convert whatever the users generate into valid commands. If many users use similar forms, then these may be candidates for adoption in the final system.

Usually, the wizard should see exactly what the user sees on his screen, plus any other useful information. Whenever errors occur, and in certain other situations, the wizard can take control and provide input directly. The wizard will also need to be able to edit unsuitable output from the system before the user sees it. The wizard will probably need to perform experiments (e.g., to find a file the user seems to be referring to) without the user being aware.

Note that the Wizard of Oz is not only a powerful development technique, but in suitable form can also be retained in a final system to enable skilled users to help their colleagues. (Imagine that one user has got stuck; he requests help from another user; that user becomes a wizard for the first and helps him out.)

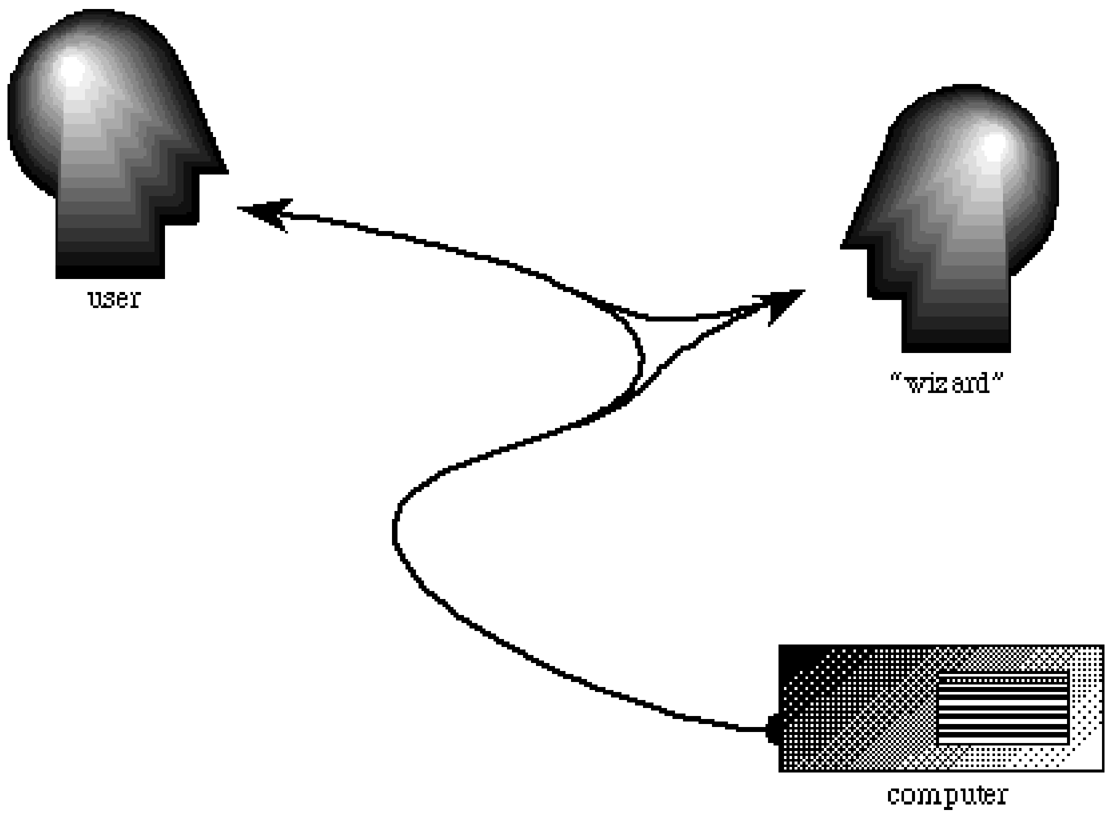

Figure 6: The Wizard of Oz

7 A simple, effective design method

Given the wide range of environments and range of user skills for which systems may have to be developed, it might appear that there would be a correspondingly large range of design methods. It is however possible to summarise a powerful method for user interface design that applies to the entire range of human-computer interfaces, from programming language designs (not very interactive; user relaxed and alert) to airplane control (real-time and highly interactive; user stressed and fatigued).

The method can be summed up as *Utter Honesty plus Occam's Razor*,[5] and is worked out as follows:

1. Document the intended system from the user's point of view. The more detailed the documentation the better. If you can't be bothered to document something, take that as an indication that perhaps the feature is not worth having anyway.

2. Describe *all* known bugs, provisos, limitations, side-effects etc. If there are known ways around the bugs ('work-arounds'), write them down too (then ask yourself why the program cannot do them for the user).

3. Improve the *system* by simplifying the *documentation.* In particular, you will improve the system by (truthfully) being able to remove the bug warnings that you entered into the documentation under Rule 2.

In practice, these steps will be iterated and interleaved with implementation and developing production documentation. It is possible to compromise the method by designing only the core of the system in the way suggested: this results in a **minimal manual**. Minimal manuals, which correctly describe a subset of the system, have been found to be very effective for user training, but to be effective they require a way to 'block off' the user from the advanced features that are not described in the minimal manual.

For example, one word processor manual warns the user that it is not possible to delete more than 100 paragraphs in one go. But to follow Rule 3, the system should now be changed so that this bug warning is no longer necessary. Maybe the system was written in Pascal with an array of 100 elements

[5]For an essay on utter honesty see [3].

. . . but surely the program could be changed so that if the user tried to delete more than 100 paragraphs, the program iterated, deleting no more than 100 paragraphs per iteration. This should be easy enough to implement, and doing so would simplify the manual, and improve the system.

Simple, complete documentation is a good thing, and making it so improves the system. In the absence of any agreed ways of evaluating success with design, the length of the documentation is a good indicator. Reduce the length of documentation, not by glossing bug warnings, but by fixing bugs in the implementation.

This three-step approach is of course facilitated by gueps (§5), particularly because of the active role of developing documentation enabled by gueps: improvements are led by the documentation, not by the computer implementaton.

8 An exercise

As an exercise (whether real or imagined) consider designing a system to play chess. Although this example has been deliberately kept simple, note:

- How many alternative design options are available. Tradeoffs are not very obvious 'in the abstract,' and almost certainly a prototype system must be built and assessed.
- How many features that are useful for design and evaluation of the system can also form useful features for the user; notice, too, that a 'simple' self-contained application like chess throws up very many creative possibilities, and that the designer has to make careful tradeoffs.
- How very soon a very detailed knowledge of the application (in this case chess) is essential to implement a satisfactory system. (Indeed, too many chess programs currently available do not correctly implement the basic rules of chess!)

Chess is a well-known application, and you can assume that users will know how to play the game (but do you want to provide tuition features?) Implementing the chess 'engine' itself is relatively straight-forward, indeed you may have access to a suitable program whose interface can be modified (either by reprogramming or by filtering the chess program's input and output).

Chess can of course be played on a board with real pieces—a Category 2 interface (§3)—but more usually you will implement a chess program on a workstation, possibly making use of graphics to provide a display of the board in photographic realism (with perspective and lighting effects?) or in symbolic form (as used in conventional printed chess boards). There are clearly a wide range of choices for the representation of the board; and there is a complex tradeoff between realism and obtaining an adequate speed. There is a subtle tradeoff that the more effort put into the interface, the more the system will exploit idiosyncracies of the hardware it is running on, yet the more the designer may wish to amortise his design effort by making the system portable across many different sorts of hardware.

If the user has few choices (perhaps only one), the system may **highlight** alternatives for him. In fact, the user may want the computer to make the best move for him, that is, to accept the default offered.

There are many alternatives for input styles. Direct manipulation is feasible: the user simply points at a piece and moves it to its destination square. All moves apart from promoting pawns (the user will normally promote to a queen, but need not do so) are strauight forwardly handled by direct manipulation.

Chess has various symbolic notations which suggest alternatives to direct manipulation. Instead of physically moving a pawn, the user might type `d4`—meaning move a piece (e.g., the pawn previously on d2) to square d4. The standard notation provides ample scope for defaults. **Command completion** is a technique where the computer, anticipating what else the user will input, completes the user's command. For example, if the user types `a` and the only piece he can move on the a file is a pawn at a3, the computer could complete the command thus: `a3-a4`, saving the user typing four characters (or beep if that piece could not be moved). Command completion can be annoying (because it doesn't always happen), so it is usual to provide a **completion** button: command completion then occurs only when this button is pressed—it amounts to accepting a default given a user-specified prefix.

An **equal opportunity** interface would allow the user to move either by direct manipulation or by standard notation. If the user moves a piece by pointing and dragging, then the computer generates the symbolic form; conversely, if the user types the symbolic form, the compter moves the piece. It will be neat to permit parts of moves to be made either way; thus, typing `a4` makes the piece of square a4 flash (just as it would if it had been selected by pointing); alternatively, selecting a piece by pointing at it, then typing `a5` moves the selected piece to square a5. In short, equal opportunity removes distinctions between input (e.g., the typed commands) and output (the changes to the board). More generally, equal opportunity suggests allowing the user and computer to exchange sides (viewing the computer's moves as its output responding to the user's moves as its input), anyway a useful feature to consider for a chess program.

An advantage of standard chess notation, and written notations in general, is that a **log** can be kept of the interaction for perusal later. Obviously in chess, a chess player is probably interested in his choice of moves that led to a certain outcome—but the designer is also interested in the log (especially if it is annotated with times and errors) to see how the user interface might be improved. The user, too, will want to annotate the log, e.g., with insights about the position, or alternative moves considered—a point confirming the view that designer tools make good user interface features for users. On a workstation in may be desirable to show the complete log of the game at all times; since this may be too large to fit, there will need to be some mechanism to **scroll** it (to move it vertically).

People often make mistakes, and although it is strictly illegal in chess, there are cases (e.g., solving set problems) where the user will want to try out moves 'to see what happens' and if the outcome is unwanted to **undo** his previous move. In general, the user should be able to undo moves right back to the start of a game, and indeed to **redo** undone moves in case he undoes too far back.

Chess is often played under time constraints. The system should display a **time to completion** indicator. In chess this need only show total time *remaining*, but in other applications it is more useful to show time remaining before the user next can reply, sometimes as a **per cent done** indicator, as shown in Figure 4. (In some games, players may be required to move within a certain agreed time per move.) It is interesting that on time-shared computers that time to completion can be estimated accurately merely by being conservative: any excess time can be spent on other tasks. The user benefits because the time to completion is reliable.

As in many applications, the user may have to suspend a game (e.g., while he goes for a cup of tea); the computer may have to be switched off or the session terminated by logging out. In all cases, then, the computer should provide a way of **saving** and **restoring** games. In general, a player may want to have more than one game in progress, so there should be a way of identifying suspended games for later recall. (This is analogous to, say, word processing where a user may be typing several documents, 'suspended' documents being called **files**, identified by name.) It is good practice to provide an integrated way to save games on alternative media, in case of hardware faults rendering the system inoperable. In particular, hardcopy is usually desirable and in the last resort the user can always retype a game by hand—a requirement which in turn requires the user to be able to specify the computer's moves.

Chess provides an obvious application of the use of the Wizard of Oz technique. Instead of playing the computer, the computer merely acts as an intermediary for another human player (perhaps hidden in another room, so that the actual user does not know the difference). The user can be his 'own' wizard if the system allows players to change sides!

Chess is a good vehicle for experimenting with the use of speech and sound, to diagnose user errors (illegal moves) and to announce moves, mate and so on. An interesting design principle would be, "a game can (in principle) be played with your eyes shut," meaning that the sounds provide enough information for the player to know exactly what the board position is. Blind or blindfold chess players might enjoy this, but most players would certainly benefit from the redundant output **confirming** what they can see happen. (If the computer makes moves using 'direct manipulation,' speech would disambiguate similar moves such as a bishop or pawn taking a piece.)

It is possible that users could be taught playing strategies by viewing, for example, end games animated at high speed. Equally designers could replay interactions at high speed to identify salient features of interaction. Colour (or gray level) could be used very effectively for indicating certain sorts of positions (e.g., forks, the 'square of the pawn,' or the combined force threatening a particular square).

Unusually for such a simple application, chess also provides interesting possibilities for programming. A user may want to play the Sicilian or other book opening, or castle, or to chase the black king into the corner. More interesting possibilities are to change the rules (e.g., to play Scotch chess) or to introduce pieces that move differently.

There are many formal interface design principles applicable to the chess application; here we list a few that are readily formalised and also expressed colloquially as 'golden rules.' Some principles seem obvious—but consider how similar rules are flaunted in many interactive applications, such as word processors or spreadsheets.

1. The system must have **levels**: each level selecting a different set of laws of its behaviour. One level certainly has to restrict behaviour to FIDE tournament rules,[6] another level allows setting up of arbitrary positions, another level allows imposing handicaps (e.g., removing a queen). And so on.
2. Although the user can change levels at will, it is reasonable to impose laws about level-changing. Thus it should not be possible to set the level to tournament rules after 'cheating' by operating at a different level! Nevertheless, the user may want a level so that the system acts as if at the tournament level.
3. All errors are notified by a beep and by an explanatory message on the screen. The message disappears when the cause of the error is rectified.
4. A distinction is made between preemptive and non-preemptive errors/warnings: at some playing levels, all errors must be preemptive (to stop cheating); at others, some errors (e.g., that a player is in check) may be because the user is constructing a position, further pieces have yet to be placed.
5. The system displays the complete **state**. That is, it must display the board, all pieces (in their proper places and colours), indicate whether castling and check have occurred for each side, time remaining for each player, move number, moves since a piece has been taken, repetition count etc. If games can be suspended, then their names (or other unique identification) must also be displayed.
6. A saved game saves all the state.
7. Every state must be **reachable** (at the appropriate playing level). Thus the user must be able to set up any position, including specifying previous castling and checks.
8. Undoing a move always causes the computer to take back the corresponding move; redoing a move causes the computer to replay the original move (and not some alternative). Thus undo-redo are inverse operators.
9. Defaults (command completion) are only suggested when there is exactly one move. (Though other playing levels may provide 'suggestions' rather than defaults.)
10. Abbreviated moves (e.g., `d4` or `Nd4` instead of `Nb2×d4`) are only accepted when unambiguous.
11. Since users make mistakes, and when the playing level permits it, there should be a way to delete typing mistakes. Typically, when the user hits `DEL`, the previous keystroke is deleted. The relevant principle, therefore, is that the system should revert to *exactly* the state it was in before that key was pressed. (Thus, any command completion should be taken back; any visual feedback on the board part way through a move showing the selected piece should revert.) A second principle (at the appropriate playing level, if any) is that there is no limit on the number of `DEL`s that can be typed ... even deleting back through previous moves. (Note: `DEL` acts at an interface level, undoing actions such as keystrokes, whereas undo acts at an application level, undoing actions such as moves.)
12. Whenever there is a delay (e.g., the computer is considering its move), the screen should show a suitable indication that something is happening. Typically, the cursor will change to a 'watch' or 'hour glass' symbol. This **wait symbol** is a special case of the percent done indicator and can be used more generally.
13. If the log is displayed and scrollable, then no changes occur to undisplayed parts of the log. Thus, when either player makes a move, the log must scroll to the appropriate position (presumably the end, unless this move is an undo or redo of earlier moves). This is a special case of the rule: when things change, the user can see them change.

[6]'FIDE rules'—you need to know what these are before implementing a chess program!

9 Summary

The user interface is often left till last, as obviously (?) the 'real,' underlying, part of the system must be implemented first. The user interface tends, then, to be very much an afterthought (often a development from the system debugger!) and the opportunities for creativity within the constraints of top-down design are lost. User interface design is no less of a challenge than programming the intricate algorithms of the application, indeed, it is more of a challenge because of the uncertainties of the user's needs and behaviour. It deserves to be designed in step with the rest of the system; the designs of the underlying application and interface mutually suggesting improvements.

This chapter raised many of the salient issues in user interface design and suggested various approaches to the effective design of interactive systems. It has not covered psychological techniques, for instance, to find out what users really do or want, or how they really go about using computer systems when nobody is watching them. Instead it has promoted principle-led design, and allowed that even with the best intentions bugs arise, and these must be acknowledged, and their acknowledgement can lead to improvements in the overall system design.

References

[1] Apple, *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley, 1987. ▷*Proprietary guidelines for developers of Macintosh applications; much more restricted, then, than [11], but collectively impose a distinctive and effective style. The emphasis is on direct manipulation interfaces.*

[2] R. W. Bailey, *Human performance engineering: A guide for system designers*, Prentice-Hall, 1982. ▷*Practically all the psychology and ergonomic information that a designer should be aware of.*

[3] R. P. Feynman, *Cargo Cult Science*, in *Surely Your're Joking, Mr. Feynman!* Bantam Books, 1985. ▷*The final chapter of this brilliant book is an appeal to do science honestly. It seems to me that a lot of computer science, particularly user interface design, would be the better for honest standards.*

[4] M. D. Harrison & H. W. Thimbleby, *Formal Methods in Human Computer Interaction*, Cambridge University Press, 1990. ▷*Summarises recent research in formal aspects of HCI, in particular bringing software engineering techniques to bear on the HCI design process. Also discusses prototyping and UIMS, etc.*

[5] M. Helander, editor, *Handbook of Human-Computer Interaction*, North-Holland, 1988. ▷*A massive guidebook to human factors (psychological/ergonomic) engineering in all aspects of user interface design; summarises both research and recommendations. Unfortunately does not discuss implementation issues.*

[6] I. Lakatos, *Proofs and Refutations*, Cambridge University Press, 1976. ▷*A play exploring and developing the creative side of mathematical proof; relevant here for its implications on the creative application of formal methods in 'even' user interface design.*

[7] D. Levy, *Computer Chess Compendium*, Batsford, 1988. ▷*A collection of computer chess articles, ranging from the classic Shannon and Turing papers on computer chess, through the de Groot, Chase and Simon psychological studies of skill in chess. Alan Turing mentions paper simulation to help design chess machines; but the only comment about user interfaces as such relates to checking for illegal moves—and is dated 1958, when punched cards were the state-of-the-art user interface!*

[8] B. Myers, *Creating User Interfaces by Demonstration*, Academic Press, 1988. ▷*A useful discussion of UIMS, particularly concentrating on the author's system, Peridot. Peridot enables interfaces to be built largely by 'demonstration.'*

[9] W. M. Newman, *Designing Integrated Systems for the Office Environment* McGraw-Hill International, 1986. ▷*Complements [10] with very good descriptions of many office systems, including networks, graphics, integrated systems.*

[10] B. Shneiderman, *Designing the User Interface*, Addison-Wesley, 1987. ▷*A good survey of the human side of interactive systems design. Intermediate between [2] and [13] in orientation towards computers.*

[11] S. L. Smith & J. N. Mosier, *Guidelines for Designing User Interface Software*, Technical Report NTIS No. A177 198, Hanscom Air Force Base, MA. ▷*A massive compendium of user interface design guidelines. This collection has been repeatedly revised in the light of new studies: more recent editions may be available.*

[12] R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall, 1981. ▷*An introduction to the design principles underlying programming languages.*

[13] H. W. Thimbleby, *The User Inteface Design Book*, Addison-Wesley, 1990. ▷*This chapter is based on this book, which is a more abstract survey of user interface design than [10], promoting formal methods and their creative potential in design.*