Many problems that have to be solved in present day human-computer interfaces arise from technology limitations, quite apart from those arising from lack of appropriate knowledge. Some of the progress we see in the most recently developed interfaces has occurred simply because bit-mapped screens, large memories, colour, compute-power appropriate to local intelligence, and the like, have all become inexpensive at the same time as rising human costs have finally been appreciated, and deprecated, by those who pay the bills. The new technical possibilities, and the now obvious economic advantages of providing good interactive computer support to enhance human productivity in all areas of endeavour has created tremendous pressure to improve the human-computer interface. This pressure, in turn, has dramatically highlighted our lack of fundamental knowledge and methodologies concerning interactive systems design, human problem solving, interaction techniques, dialogue prototyping and management, and system evaluation. The design of human computer interfaces is still more of an art than a science. Furthermore, the knowledge and methodologies that do exist often turn out to fall short of what is needed to match computer methods or to serve as a basis for detailed algorithm design.

The paper is addressed to a mixed audience, with the purpose of reviewing the background and current state of human-computer interaction, touching on the social and ethical responsibility of the designer, and picking out some of the central ideas that seem likely to shape the development of interaction and interface design in future computer systems. Areas are suggested in which advances in fundamental knowledge and in our understanding of how to apply that knowledge seem to be needed to support interaction in future computer systems. Such systems are seen as having their roots in the visionary work of Sutherland (1963), Englebart and English (1968), Kay (1969), Winograd (1970), Hansen (1971), Papert (1973), Foley and Wallace (1974), and D. C. Smith (1975). Their emphasis on natural dialogue, ease of use for the task, creativity, problem solving, appropriate division of labour, and powerful machine help available in the user's terms will still be crucial in the future: However, the ability to form, communicate, manipulate and use models effectively
will come to dominate interaction with future computer systems as the focus of interactive systems shifts to knowledge-based performance. Human-computer interaction must be regarded as the amplification of an individual's intellectual productivity by graceful determination and satisfaction of every need that is amenable to algorithmic solution, without any disturbance of the overall problem-solving process.

1. Introduction

1.1. A prospectus

A non-technical vision of the future possibilities for human interaction with computers has been provided in a variety of media including several recent movies. The story that really centered on this interaction and interplay was that involving HAL, the shipboard control computer for a voyage to Jupiter, following the summons of an alien intelligence (2001: A Space Odyssey, by Arthur C. Clarke). More technical views have been provided, at least in part, by developments in the field, documented in the technical literature, but on a piecemeal, scattered basis. Two recent surveys of directions in human-computer interaction concentrate on the application of Artificial Intelligence (AI) to interactive interfaces (Rissland 1984, Vickery 1984) and highlight the increasingly important role seen for AI in future human-computer interaction. The Architecture Machine Group (AMG) project, which has been underway at MIT since 1976, provides one of the more ambitious non-fictional views of future interaction. It is based on the exploitation of spatiality and other normal properties of evolved human perceptual motor performance in a computer-simulated ‘Dataland,’ and is intended to complement more conventional forms of interaction (Bolt 1979, 1980, 1982, 1984). However, HAL serves as an important different view of possible integrated interfaces of the future, all the more powerful because the view is set in the context of a real task, but forms the background and plausible context for action, rather than being the focus. As in the past (with submarines, space flight, and the weapons of war) art suggests and defines the future goals of our technology.

1.2. Why better interfaces?

In the last year or two, there has been an upsurge of interest in providing better ways for people to interact with information processing systems. There are at least two reasons for this. First, it has become apparent that poor interfaces make it more difficult for users of computer systems (including computer science experts) to do their job. Better interfaces improve productivity, reduce errors, and allow higher quality results. They give a competitive edge to their suppliers and, incidentally, make the users more comfortable in their work. With falling hardware costs and rising labour costs, the emphasis has changed from utilizing machines to their maximum capacity to utilizing their human users and operators to best
effect. For once, this is a trend that also benefits these people directly.

Secondly, computers are becoming very widely used, even in areas and in equipment that have previously not been associated with computers. The users of computers, in these circumstances, frequently have little or no computer training and, collectively, may exhibit the whole gamut of educational and career achievement in their various specialities. For such people, the computer should appear as a tool, interfaced in such a way that the user can think about the task goals for which the system is used, rather than the characteristics of the computer tool used to achieve these goals. Some systems must carry the computer power so deeply embedded that it is effectively hidden, just as the electric motor in a dishwasher or clock is hidden. The interface seen by the user is completely task-oriented, and the internal logic of the system (programmed, even in the case of non-computer equipment these days) translates the user’s needs into the control and/or power signals required to employ the technology as a subsystem. Of course, the user may well be aware that a computer (or motor) is in there doing essential things, but does not have to be concerned with its characteristics.¹

Thus, so-called *user friendly interfaces* have become the touchstone for the more widespread and effective use of computer power. Such interfaces have a direct economic and social impact, to the extent they succeed or fail. They allow the computer industry generally to expand markets, hence creating new jobs within the computer industry. Good interfaces also allow other companies that use the new computer power to be more productive and competitive, which may not only expand their existing market shares but also lead to new markets for information technology in previously untouched application areas. There is a warning here for those societies that feel they can remain as mere users of the new technology. Future markets will increasingly deal in the products of the new information technology industry, with employment in traditional areas declining as the new machines make the remaining employees more productive. Balance of payments problems will explode for those countries that face the need to import the new technology to remain competitive, through failure to develop it themselves.

A few years ago the graphics area in computer science expanded dramatically as the need, the methodology, and the technology appeared or were generated. Advertising, film-

¹ The analogy to embedded motors was first suggested by Weizenbaum 1975).
making, and design have provided much of the finance and incentive to the graphics expansion. Now that costs have fallen (as research has been amortized, as mass-market software has been developed, and as mass-produced hardware tailored to the specific needs of computer graphics has started to appear), computer graphics is providing part of the base for better human-computer interface design. Other technologies are starting to mature: expert systems; low-cost very powerful desktop computers with high-resolution colour displays; dialogue prototyping and management systems; databases and database access methods (especially limited natural-language-based access); new kinds of input-output devices that are also inexpensive (speech input-output devices, innovative direct manipulation media, etc.); and so on. It is now commonplace to do things that were not possible even as recently as two years ago. Not only does this allow new approaches to human-computer interfacing but it also allows sophisticated interfaces to be created quickly and at low cost. This, in turn, facilitates better and more diverse experimentation related to human-computer interaction, as part of the research needed to expand the body of knowledge concerning the methods and goals of human-computer interface practice.

The Apple Macintosh, developed from the Lisa (Williams 1983, Morgan et al. 1983) is an example of a current popular application of both new technology and new knowledge. The technology and experience that made this approach to computing possible has its roots in the visionary work of Sutherland (1963) who invented the first ‘graphics-land’, with elegant graphical interaction techniques, employing unobtrusive machine assistance, to amplify the drawing skills of the draughts-person unconcerned with the technicalities of computers; of Englebart (1968), who originated the mouse and computer-augmented human reasoning at SRI; of Kay (1969, 1972) who developed the first higher-level personal computer, object-oriented programming with windows and multiple views, systems based on message-passing primitives, and simple personal programming systems of great power; of Papert (1973, 1980) who, following in the traditions of Piaget and Montessori, used computers to show how complex ideas could be taught easily when translated into concrete terms in an environment in which it was easy and enjoyable to experiment, catering to the growth of the child rather than mere provision of information; of Foley and Wallace (1974), who made a notable early statement of rules for natural graphical ‘conversation’; and of D. C. Smith (1975) who developed direct manipulation and the ‘icon’ as the basis for computer-aided thought using ‘visualization’, inspired by the
visual simulations and animations of Smalltalk, Kay’s system. But the Macintosh would not have been possible as a popular personal computer without technological advances in microchip design and fabrication, allowing cheap memory and processing power as a basis for bit-mapped graphics, speed, and powerful interactive software. Now we have the Atari 1040 ST that offers similar facilities not for US$2500, but for US$900, and the Commodore Amiga at US$1200, both with higher resolution and excellent expansion capabilities.

In the face of this technological cornucopia coupled with an abundance of relevant ideas, it is becoming increasingly clear that interface design is still an art, and that art is being severely taxed as the purely technological limitations disappear and as an increasingly large number of would-be users are able to afford the hardware to support their activities. The remainder of this paper leads up to a discussion of themes and ideas that will be important in interacting with future computer systems (in Section 6). In preparation for this, three important issues are addressed: (a) the ethical and practical constraints on the application of future computers, since these form the context and rationale for interaction; (b) the distinction between programmers and users, and the nature of the programming task, since programming is an important form of interaction with computers; and (c) the game element in human-computer interaction, because evidence suggests it may be possible to improve interfaces by exploiting some features of games. In Section 5, a futuristic database access system (Rabbit, Williams 1984) is described, because it begins to incorporate ideas that seem crucial in future computer systems interfaces. Finally, there is the discussion. The central theme in future human-computer interaction will be the formation, representation, communication, manipulation and use of models. Other important themes comprise redundant, multi-modal interaction techniques; and the specification and management of interaction. These are addressed.

2. A context for future interactive systems

2.1. Introduction: the ‘do it’, or abdication model of interaction

The easiest way to get something done is to ask a competent loyal assistant or colleague to do it for you or, if your involvement is necessary, to assist you in doing it. Given appropriate talent, this may be even more effective than doing it yourself. The metaphor has been used before in the context of a programmer’s assistant (Teitelman 1972, 1977), and tends towards one extreme in the continuum of views of the user interface. This extreme looks for an active, intelligent, reasoning mediator that lies between the user and what is to be done. The other regards the interface as a simple passive ‘gateway’ or membrane between a user and the application (Rissland 1984), that can be tailored to particular needs, perhaps, but is simply
a personalizable tool—not even a good servant, let alone an assistant or colleague. The issues involved are: where is control located; and how much expertise can be built into the interface management? However, these questions are bound up with questions about the structure of User Interface Management Systems (UIMSs) and about task allocation in humancomputer systems—why is it necessary for humans and computers to co-operate?

If we naively assume that the interface for future computer systems will comprise a voice input natural language ‘Check this task definition and do what I mean’ (DWIM) command system, perhaps with graphical aids to help in task definition, we are overlooking certain fundamental facts concerning the reasons for using computers, as well as both current and absolute limitations on their use. We are also underestimating the problems involved in communicating with colleagues and assistants. True, there will undoubtedly be an increasing number of tasks for which the relevant experience and applicability criteria can be defined to allow something approaching this type of interaction. However, even these systems, which already exist in embryonic form, must place a certain emphasis on the metacommentary aspects of the dialogue involved, and respond to questions or comments related to their internal workings and dialogue construction, as well as to specific task goal elements. Control ultimately resides with the human, and human goals must be satisfied. Issues of communication and metacommentary, as well as the effect of conflict between the goals of the communicants, are nicely summarized in the work of Thomas and Carroll (Thomas 1978, Thomas and Carroll 1981).

The problems of task expression and refinement, of knowledge acquisition and retrieval, of reasoning and planning, and of goal resolution make the DWIM type of interface a remote dream as a general-purpose form of interaction with computers. It was this kind of interface that was portrayed in the movie 2001: a Space Odyssey, and the system ultimately broke down due to the conflict of goals at several levels. Moreover, in the end, the computer was obliged to lie in answer to questions at the metacommentary level, and finally to attempt to take over absolute control, a strategy only thwarted by the creative problem-solving performance of the remaining crew member. Rasmussen (1983) refers to this kind of performance as knowledge-based performance, as opposed to rule-based performance in which situation-action rules are remembered from previous experience, selected as appropriate, and applied. Weizenbaum (1975) has argued forcefully against the belief that, in principle, computers may be able to take over the running of society completely, and his arguments bear on the
topic raised above concerning absolute limits on what computers should do. Since his view requires human involvement in certain kinds of activity, it also requires human-computer interaction, no matter how sophisticated our computer systems become. There is of course the question, if HAL-like interaction is feasible, is it desirable or even preferable to more traditional forms?

Society’s reaction to the modest progress in applying computers alarms Weizenbaum, who sees the potential for dehumanization, inflexibility, control, and oversimplification inherent in the unwise and over-hasty application of computers in areas we either do not understand well enough, or from which we should exclude computers for ethical reasons.

Much of the force of Weizenbaum’s case derives from arguments about the level of understanding required to model situations or systems as a basis for solving problems, and from arguments about our ability (or, more likely, lack of ability) to implement such models as computer programs. Both sets of arguments centre on problems created by the complexity involved, as well as the character of the entities being modeled. These, in turn, affect the questions we ask, can ask, or should ask in order to formulate the model in the first place. From this ground, Weizenbaum argues that computers are being applied in harmful ways for a variety of derivative reasons. First, inflexible solutions to problems are created because complex programs—especially those written by a team—are themselves not understood well enough to permit changes to them, even to correct known errors. Secondly, solutions are based on incomplete models and data, due to our lack of understanding, and our lack of ability to formulate adequate questions to illuminate even those aspects we are aware of; let alone all the questions that we should ask, if we had God-like insight. There is also the question as to whether all relevant matters could be covered by such a factual approach. As riders to this, Weizenbaum points out that (a) data may be ignored simply because ‘it is not in the right form’, and (b) oversimplified solutions will be produced based only upon those aspects of the problems that we can formalize. A third harmful effect of computers, he argues, is that they act as a conservative force in society, partly by providing the means of sustaining outdated methods of running an increasingly complex society, and partly because, once programs are written, they are so resistant to change, for practical as well as economic reasons. Finally, he argues that computers have made society more vulnerable. With continued centralization of control (such centralization itself outdated), errors and disturbances have far-flung and unpredictable consequences as they propagate through a
homogeneous system, optimized for economy rather than stability. The scheduling of airline flights is an example of such a system in which unplanned hijacking incidents have propagated their dislocating effects on a world-wide scale, by domino action in a system with inadequate flexibility. Equally, the recent mini-crash of the Wall Street stock market (September 1986), which rippled around the world, is attributed by experts to slavish adherence to predictions and recommendations generated by computer models of stock market performance that were inflexible and incomplete. It is also increasingly obvious that, as in all human activity, economic considerations tend to act in such a way as to simplify solutions and to inhibit improvements that cannot be proved to bring directly measurable financial or political benefits. Such attitudes are much harder to attack when entombed in the amber of computer software.

Alongside this technical theme to Weizenbaum's book, there runs a strong philosophical argument against the dehumanization of life and society. The most important point is this: by insisting that logical solutions to problems are equivalent to rational solutions to problems, one is defining out of existence the possibility of conflicting human values, and hence the human values themselves. Here can be seen the basis of conflict with many researchers in Artificial Intelligence, for the whole philosophical thrust of the book is against the view that the human being is just a computer, with mechanisms and rules that can be understood and transferred to a machine. In Rasmussen's terms, computers may assume a major share of the performance burden at the rule-based level, minimizing and simplifying the interaction required in the process, but the real challenge for future computer systems will be to facilitate human-computer interaction in a knowledge-based performance mode. If Weizenbaum is right, and I believe he is, knowledge-based performance can never be completely taken over by the computer because it is neither possible, nor ethical. The computer must remain a smart tool in the search for formalizations of useful new knowledge, or of new insights into old knowledge, conditioned by the goals and needs of humans. However, real progress is possible, in terms of the acquisition and application of knowledge, if we can solve the problems associated with the formation, representation, communication, manipulation and use of models in interactive problem-solving and task execution. In this way, the ethical and practical objections can be overcome, whilst still maximizing the

1 Webster defines rational as having reason or understanding; being reasonable; whilst logical means formally true. Logic is, ultimately, tautologous, and denies conflict. By denying conflicting human values, logic, in essence, denies the reality of the values themselves.
support to the human. This is why human-computer interaction will be so important in future computer systems, and why models will feature so prominently and importantly. Shared models will form the knowledge interface between computers and people.

There are really two kinds of question raised by Weizenbaum’s book. One kind is technically oriented; questions about the best division of labour in a system involving both humans and computers; questions about the practicality, validity and utility of partial solutions to problems we do not fully understand; and questions about the state of our knowledge concerning how to implement certain kinds of solutions adequately. There is also the question as to whether some kinds of problems are amenable to programmed solution at all. These are all valid research questions that cannot be ignored as we design increasingly complex systems. We should not get carried away by the modest success in improving knowledge access that has been achieved on the basis of rule-based ‘expert systems’.

The other kind of question begs the reader to step outside the conventional framework of disinterested science and ask questions about the value and ethics of what is being done with computers in terms of replacing people and running society. The underlying, but unstated message here seems to be that, if we are approaching God-like powers with our technology, we need God-like wisdom and restraint in the exercise of these powers. The implication is that the only viable basis for restraint and wisdom, on the scale required, is for each individual in the technological and scientific areas concerned to take some personal responsibility for the consequences of exercising his or her professional skills. This is the context within which we should contemplate the creation of future computer systems, and the context which constrains the character of our interaction with them. This is why the human-computer interface will grow more complex and demanding, rather than less, as our knowledge increases. Understanding such interfaces becomes tantamount to understanding ourselves, yet considerable understanding is required as a basis for design.

If the user must continue to be an active participant in increasingly sophisticated future computer systems, which is the logical and ethical conclusion from the foregoing discussion, then the human-computer interface is not only here to stay, but must develop appropriately. Furthermore, whatever the status of the user as a computer specialist, the user must have some task-relevant knowledge. It is the unification of the two sources of knowledge, human and computer, in the
problem solution, that is the ultimate goal of humancomputer interaction. Williams (1984) points out that the knowledge brought to the task by humans very likely differs from that brought by machines. That brought by the human is high-level generic knowledge whilst that brought by the machine is the lower-level, physical-particulars kind of knowledge. Again in Rasmussen’s terms, the human tends to have a model at an intentional level of ends, whilst the machine is able to provide models at the physical level of means. The mapping between them is many to one in both directions. Interaction applies the means to the ends by forming or invoking particular functional models that connect the two. For this to work, mechanisms must be available to allow the participants to explain themselves to each other and form the connections. Furthermore, any such process should result in the creation of, or accomplishment of, something relatively perfect and formally correct (the solution to the original problem) from an error-prone sketchy interaction. Interaction must be regarded as amplifying an individual’s intellectual productivity by graceful determination and satisfaction of every need that is amenable to algorithmic solution, without disruption of the overall, usually knowledge-based performance of the human. The key to this is the effortless sharing of the models that embody the various kinds of knowledge involved: their formation, representation, communication, manipulation and use. Where those models are partially or completely inaccessible behind the human cognitive veil, for whatever reason, then the interface must support the elicitation and communication of incomplete constructs and informal descriptions based on the results of using those models covertly.

Experience has shown that good interfaces make it easier for computer users to do their job. Even computer experts show increased productivity, reduced errors, and higher quality work when they are provided with a better programming environment and more powerful tools that are easy to apply to their work. Furthermore, falling hardware costs and rising labour costs are shifting the emphasis from machine utilization to human productivity, in terms of increased throughput, reduced errors, shorter training periods, and lower staff turnover, whilst still maintaining or preferably improving the quality of work produced. With the increasingly widespread use of computers by non-experts for a variety of economic and practical reasons, this situation (as already noted) has led to a dramatic surge in the attention given to the human-computer interface in applications areas. (Unfortunately, when advertising products, all too often the attention is mere lip service.) However, the corresponding rise in our knowledge of how to

3. The programmer as a user

3.1. Current support for programmers
design good interfaces, even for well-defined applications tasks, has been far less than dramatic, again as noted.

Surprisingly little has been achieved in terms of providing good supportive interfaces for programmers (programming environments) despite the fact that, in a very real sense, one of the most important applications of computers is to programming. Not that the problem has been ignored by researchers. There have been studies and experiments concerned with various aspects of the psychology of programming and much written about the value of structured program design and the relative merits of various kinds and levels of languages. Problems of specification, program comprehension and debugging have been considered. Curtis (1981) provides a useful selection of papers up to 1981 but, for example, in the classification system for human-computer interaction literature appearing in the special issue of *Ergonomics Abstracts* devoted to human-computer interaction (Megaw and Lloyd 1984), the word *programming* (or anything like it) appears only twice. Even then it is only in connection with languages, and with ‘aspects’ which turn out to be mostly the psychology of programming. There has apparently been little success in integrating some of the available knowledge into programmer interfaces (programming environmentsystems or applications oriented) comparable to those available for end users. The sum total seems to be a collection of fourth-generation tools to assist in screen management and *Unix*. Programmers are still largely left to look after themselves, which may boost their egos, but hardly boosts their productivity or the quality of their products.

3.2. The programmer’s needs

A more comprehensive approach to meeting the programmer’s needs seems reasonable. Thus, a future programmer’s environment should allow different parts of programs to be implemented in whatever languages are appropriate, and run on arbitrary machines in a distributed system according to the best match between algorithm and machine, the latter without requiring any intervention by the programmer or (if there is one) the end-user. This requires smooth, language-independent module interfaces. Debugging tools should understand a lot about program structure and behaviour, as well as about data structures and how they are used, providing a higher level interface together with expert help for the programmer looking for faults. Structure editors (DonzeauGouge *et al.* 1975, Neal 1980) embodying the syntax and character of any programming language or document in use should be available. File systems present a particular problem and probably require progress in expert file management to help the user (programmer) manage and retrieve files. The
spectacle of a productive programmer searching an extensive hierarchical file structure for a lost file of uncertain appellation is sad to see. An integrated applications programming environment would, in addition, place the computational tools needed to support an application at the same level as the interaction tools needed to support the user, with control residing at a task management level integrated within the operating system that allowed the programmer to concentrate on goals, functions and solution strategies rather than mechanisms and housekeeping.

A computer user (including a programmer) thinks and/or learns about the solution to a problem with computer assistance. Papert (1980) believes the effect of computers on thinking and learning to be comparable to that of the invention of writing. He notes one important effect of using a word processor is to free the writer from tedious housekeeping, and the laborious use of writing implements, perhaps in a very unskilled manner. For children, the effect is dramatic:

‘For most children rewriting a text is so laborious that the first draft is the final copy and the skill of rereading with a critical eye is never acquired. This changes dramatically when children have access to computers capable of manipulating texts. The first draft is composed at the keyboard. Corrections are made easily. The current copy is always neat and tidy. I have seen a child move from total rejection of writing to an intense involvement (accompanied by rapid improvement of quality) within a few weeks of beginning to write with a computer.’

Other specialized aids are available (spelling checkers, formatters and the like) so that productivity and quality both rise at the same time as the task becomes more rewarding, and attention is focussed on content and strategy rather than mechanism. Programmers require an analogous environment that goes beyond mere text manipulation (although, as suggested below, one of the two principal activities in programming may be documentation in one form or another).

Sheil (1981) presents an interesting view of the current state of knowledge concerning programming environments:

‘Most innovations in programming languages and methodology are motivated by a belief that they will improve the performance of the programmers who use them. Although such claims are usually advanced informally, there is a growing body of research which attempts to verify them by controlled observation of programmers’ behaviour. Surprisingly, these studies have found few clear effects of changes in either programming notation or practice. Less surprisingly, the computing community has paid relatively little attention to these results.’

He goes on in the paper to suggest that the problem is due to the unsophisticated experimental techniques used, and a
shallow view of the nature of programming skill (emphasis added).

Thus, no systematic study seems to have been made of the overall needs of the programmer as a user of computers. It seems to be assumed that the programmer is so expert that he or she (a) can take care of him or herself, and (b) has needs so arcane that they are beyond study. The ‘programmer’s operating system,’ Unix (Ritchie and Thompson 1974, 1978), is a fruit of this attitude, or possibly a caution against it, depending on your viewpoint. The original version was written by a programmer (Ken Thompson) working at Bell Laboratories strictly for himself, because he had been given a minicomputer with an operating system that did not meet his needs (Mcllroy, Pinson and Tague 1978). Subsequently, other programmers liked it so much that it spread, was adopted by AT&T, then licensed to universities (with some very expensive commercial licensing), and finally (quite recently) turned into a ‘product’. In many ways Unix represents a distillation of what serious computer scientists (who are all programmers?) need. It is usually touted as the best available. At the same time, the user interface has many of the shortcomings of other applications devised by programmers, as Norman (1981) has pointed out. Furthermore, it is not so much a system, as a playground of Dungeons-and-Dragons-like complexity, even to the helpful gnomes and hidden traps. The knowledge in such a system is contingent: that is, uncertain, accidental, and subject to the caprice of the designer. If something is not known, it usually cannot be inferred. Often the on-line manual itself is a maze of considerable extent, and is most easily accessed using some of the knowledge being sought, leading to interactive deadlock. If it is regarded as a help system, there is little help with help itself. In these circumstances, it is best to ask someone who ‘knows,’ which is why programmers learn best as part of an active community, and may partly explain the importance of embedded mail systems. Computer manuals are in any case ill-designed for anything but reference, a topic we return to shortly.

3.3. Self-help in programming

Programmers are not necessarily expert in the techniques and pitfalls of good interface design, and they may not even have a very clear idea of what they do when programming, let alone the ‘best’ way to do it (as has been confirmed by our experience (Hill 1985). In this, they are typical users, albeit with a great deal of computer-oriented expertise, including experience of coping with existing facilities. One early programming environment, the ‘Programmer’s Workbench,’ tackled the problem in a pragmatic, ad hoc manner (Dolotta, Haight & Mashey
1978), noting amongst other things the importance of documentation, and we have seen the development of Lisp and Smalltalk programming environments. But these developments only seem to constitute minor gains on existing programming techniques, with great reliance on the programmer’s expertise, as suggested above.

It is partly a lack of understanding of the programmer’s task that leads computer science research funding agencies and university administrations to veto the idea of providing certain kinds of facilities (e.g. laser printers and document preparation tools). It is possible that the only important thing that both systems and applications programmers do is to document. It is just that some of the documentation (on how to solve a particular problem) can be directly interpreted and executed by a computer. However, such ‘documentation’ is very demanding, and requires many special aids, techniques and system facilities to make it easy to produce excellent creations efficiently. Of course, the underlying activity is problemsolving—a common professional regime, but the thing that distinguishes programming is the form and context of the documentation.

Programmers also have to document at levels other than that of problem solution. This is either (a) so that other people (usually also programmers) can understand (and therefore check, debug, or modify) the problem solution statement, or (b) so that other people (often non-programmers) can use the problem solution itself in their work. This latter case involves an operating manual (or so-called ‘user manual’). Such manuals themselves often fall far short of the needs of their readers, concentrating on the formal description of the system (what it is) rather than the functional description (what it does), and certainly never venturing into the intentional description (why it is what it is, and why it does what it does). All three levels are needed by the human user, as a basis for forming the different levels of model needed to understand and therefore use the system effectively (Rasmussen 1983). The Unix manual should be no exception.

Most programmers seem to hate documentation, except the kind fed into the computer, so one obvious component of a programming system is something to make excellent program documentation easy. The difficulty of producing documentation is a problem important enough for Bell Laboratories to have produced a special environment aimed at documentation management (Fraser 1983), but it is aimed not at the overall problem (which would include programming) but rather at the evaluation of specialist technical writing.
Proposed solutions to problems rarely work as originally formulated. In this, programming is like other human endeavours. People succeed to the extent that they can debug their ideas. Proper debugging aids that help in finding and correcting the many varieties of program faults are therefore also important. Debugging is the second major activity for a programmer. We have hardly progressed beyond the kinds of systems developed in the late 1960s (Digital Equipment Corporation’s DDT or RAID (Petit 1970), despite the higher-level debugging techniques of the *Lisp* and *Smalltalk* environments. Some research is in progress (e.g. Johnson and Soloway 1983) that should bring us closer to meeting the programmer’s debugging needs, outlined earlier (subsection 3.2), but clearly an expert debugging aid is needed that can understand high-level aspects of recipes for problem solution (programs) and communicate in terms of intentions, causes and effects.

The important case of the *programmer as a user* is currently dealt with least effectively, partly for lack of knowledge, and partly for lack of effective application of what is available. More seriously, too many programmers (who seem to associate the term ‘user friendly’ with patronising, anthropomorphic, or inefficient interaction) are quite content with—indeed seem to prefer—this outcome, which hardly encourages progress. They certainly distinguish themselves sharply from mere ‘users’, even whilst claiming to know (without much investigation) what a user needs. With the increasingly sophisticated tools provided to allow ‘end-users’ to develop their own applications, coupled with the increasing sophistication of these end-users, the attitude is at best self-defeating. Incidentally, ‘end-user’ is the subclass of user that is properly contrasted with ‘programmer’. Both are users, as argued above, but possibly the distinction will not survive indefinitely. Smith’s view (introduced in the next section) certainly suggests that it will not.

In his thesis D. C. Smith (1975) is concerned with the attempt to define and create the ‘ideal’ programming environment, which means an environment in which the programmer can easily be creative, productive and correct. In the process, he ventures a redefinition of the term ‘programmer’. He poses five questions related to programming that really get to the heart of the matter for all interaction. The five questions, followed by some of his related comments, are:

1. Why is programming a tedious process?
2. What are the relationships between creating a solution to a problem and creating a problem to find a solution?
Future Computing Systems

(3) Do programming languages stimulate or inhibit creative solutions?

(4) Does creativity in art and mathematics provide any guidelines for creative activity on a computer?

(5) Can a programming environment be constructed to stimulate creative thought? What would be its characteristics?

Programming need not be tedious. The rest of this [thesis] is devoted to programming systems which make programming fun. As we have seen, creativity is an emotional process, and joy is one of its strongest emotions. There is playfulness in creativity ....

... PYGMALION brings art into computer science. Rather than providing a computer resource which artists can use to create (paint, compose music, etc.), PYGMALION is a first attempt to provide an artistic resource which computer scientists can use to create. In fact I hope PYGMALION will contribute to a reevaluation of what a “computer scientist” is. In my view, a computer scientist is anyone who knows how to do something and wants to use the computer in doing it. The view that only highly trained programmers can implement a task on the computer is intellectual snobbery of the worst kind.

... [quoting Kay] ‘The skilled programmer is necessary only because the distance between the computer implementation of the task and a person’s mental conception of it is too great.’

Rasmussen (1983), who developed amongst other things a hierarchical model encompassing knowledge and abstraction (see below), would certainly agree with the formulation of that last point. Also, the vital role of play in creative learning activity underpins much of what Papert seeks to do with computers.

4.2 Direct manipulation

It is worth noting that the underlying idea of direct manipulation is quite complementary to programming. In programming (which is symbolic manipulation) a recipe for action is generated by the user, and executed automatically by the computer. By way of illustration, consider what is needed to move the cursor, replace some characters, and check the result in the line editor supplied by Digital Equipment Corporation as part of their RT-11 operating system. One might type the following:

```
a13gdelete.me$d-9iinsert.this$0j1$$
```

which advances 13 lines, finds the text ‘delete.me’, erases it, substitutes ‘insert.this’, moves the cursor to the beginning of the line and lists it. One moves continually in an out of command mode. Omitting an Escape key, echoed as ‘S’, causes any following commands to be treated as text instead. There are many other ways of making errors—in counting lines or characters, for example. With direct manipulation, one moves the cursor to the desired location, watches the deletion and
insertion take place, and there is never any doubt about what is going on. Mistyped text can be corrected immediately. Of course, if the same (or closely similar) operation were to be repeated many times, the symbolic manipulation (programming) approach might be preferable. Once debugged, perhaps with a varying parameter or two, the operation could be reliably applied automatically. Direct manipulation requires constant monitoring and effort on each application. But for varied, one-off transaction sequences, such as occur increasingly as the majority of human-computer interaction situations, direct manipulation offers overwhelming advantages. In any case, programmed sequences or macros can be included, allowing the user to have the cake and eat it as well. The ability to operate directly, without planning and with low probability of error, is what makes direct manipulation attractive, and is part of what Smith was concerned with. Ideally, for a ‘mixed’ system, the symbolic representation of a direct manipulation action could be determined automatically, if repeated application were desired. This would effectively incorporate a form of *programming by example* (Halbert 1984) or (in a suitable context) *query by example* (Zloof 1977). In practice, several problems in implementing such a facility are still unsolved, and Halbert’s system involves automatic acquisition of tedious ‘inline’ code from direct manipulation activity with the subsequent addition of structure and variables by an editing process. However, if symbolic specification versus direct manipulation is part of what forms the supposed programmer/user distinction, such a facility would act to reduce it, whilst meeting the more general notion of catering to user preferences.

4.3. Games

The specific issue of playful enjoyable activity as a desirable metaphor for arbitrary human-computer interaction (as opposed to just education) was raised at the first conference devoted specifically to human factors in computer systems, at Gaithersburg, Maryland, in 1982. A paper by Malone (1982) was entitled ‘Heuristics for designing enjoyable user interfaces: lessons from computer games’. In it he addressed two questions. Why are computer games so captivating? And how can the features that make computer games captivating be used to make other interfaces interesting and enjoyable to use? Part of the answer in Malone’s study was that an element of fantasy was more important than simple feedback. The choice of fantasy seemed subject to audience characteristics (specifically sex differences in one of the experiments). Malone analysed motivating characteristics under three headings (*challenge, fantasy, and curiosity*) summarizing the factors found important in each, and also distinguished between toy-systems (which
are used for their own sake and involve *intrinsically* determined goals, as in games) and tools-systems in which the goals are *extrinsically* determined.

This distinction in the character of goal determination is a possible basis for separating games from real life. Of course, another important distinction lies in the fact that, in games, devastating results are only *denoted.*\(^3\) In real situations the devastation occurs, and one cannot come back to life or undo the delivery of a missile. Research by Kahneman and Tversky (1982) suggests that games may ‘hold’ players by a so-called *regret* factor. The regret factor relates to the ratio between the devastation caused and the triviality of the action causing the devastation (which may be a loss of score). Apparently people are more highly motivated to ‘try again’ the higher the regret factor (which increases as the loss increases or the action becomes more trivial). It might also be called the ‘if only’ factor. Real applications will only appeal in game-like terms to the extent that devastating results are easily avoided or brushed aside.

One rather insightful comment on games relates back to the attitude of the programmer and my suggestion that *Unix* is like *Dungeons and Dragons.* After quoting Nolan Bushnell, founder of Atari, as saying ‘A good game should be easy to learn, but difficult to master’ Malone goes on to say:

‘A good tool, on the other hand, should be both easy to learn and easy to master .... the tool users should be able to focus most of their attention on the uncertain external goal, not on the use of the tool itself. ... This distinction helps explain why some users of complex systems may enjoy mastering tools that are extremely difficult to use. To extent that these users are treating the systems as toys rather than tools, the difficulty increases the challenge and therefore the pleasure of using the system:

Perhaps this is the secret of the programmers’ resistance to change. They are so expert, that anything less challenging than an unfriendly system would not be fun.

Goals, coupled with uncertainty, layered complexity and performance feedback (scores) provide *challenge.* A system should present an appropriate level of difficulty commensurate with its power, and with the skill of the user. The layers of complexity simply allow the user to increase the challenge to keep pace with developing skill. This is a refinement of the more general dialogue design principle of aiming for ‘optimum stress’ (see Appendix B). ‘Too easy’ is boring and probably not powerful enough. ‘Too hard’ is a barrier to use, regardless of power. If skill is increasing (as it does in a game situation, or for a regular user of a system) the task difficulty can be

---

\(^3\) I am indebted to Harold Thimbleby for this insight.
increased or more complex tools made available (or discovered), to maintain the challenge. This model underlies many successful teaching strategies. Layering complexity could act not only as a means of providing increasing challenge, but also, perhaps, as an answer to the age-old dispute between simple systems for novice users and powerful systems for experts.

One suggestion for introducing a game/challenge perspective into programming came over the Unix network a year or two ago. It recognized the character of Unix by proposing that Zork (an adventure game like Dungeons and Dragons) would provide an excellent substitute for the normal shell (the operating system interface), managing files, gaining resources, printing and the like (see Appendix A). Such an interface would clearly satisfy those programmers looking for a useful toy (not unreasonable in the context of Smith’s view). Zork is also based on fantasy.

Fantasies should be emotionally appealing and expressed in terms of familiar metaphors. A system with fantasy evokes mental images of physical objects or social situations that are not actually present. Fantasy maintains interest, and hence motivation. Fantasy may also invoke useful analogies for action (thus Zork does tell us something about Unix!).

Finally, curiosity satisfaction involves a complex of factors including interesting but manageable and appropriate presentation components, randomness, and humour, as well as assistance in the structuring and extension of the user’s knowledge when appropriate.

There are, of course, some more mundane principles (not unrelated) for making computers more pleasurable and effective in use. Two of the earliest papers on interactive system design per se (Foley & Wallace 1974, Hansen 1971) are still amongst the best introductions to the basic problems and philosophy. Norman (1983) has produced an excellent restatement of the issues, while MacGuire (1982) provides a concise survey of several earlier views. Much can be learned from such accounts, but, as noted, there is no established design procedure for the human-computer interface. Appendix B provides a summary of this author’s view of reasonable design principles for human-computer interaction within a framework of the human and machine characteristics involved, with some attempt to instantiate the principles in something closer to rules. Rules and principles differ. Principles are intentional or functional, whilst rules result in specific implementation particulars. This can be illustrated by analogy with bridge design. There might be two principles: (a) that all bridges should last indefinitely; and (b) that bridges should blend into
the environment. Specific rules would then relate foundation materials to soil acidity, and paint colour to terrain type. Recognition and application of rules should be largely automatic, specifying the physical particulars of how to do something, whereas principles can be hard to interpret in given situations, and in the absence of rules. Even the recent comprehensive 448-page report from the Mitre Corporation (Smith and Mosier 1984) comprises detailed principles, rather than quantitative specification related to particular needs and goals (i.e. rules), and requires considerable expertise and insight to apply.

Following the models of human performance described by Rasmussen (1983) it is suggested that principles require the designer to indulge in knowledge-based performance, whilst rules supply the data for rule-based performance. Knowledge-based performance involves the search for, and validation of, new methods in the absence of rules. It is an open-ended search commitment, involving planning, and the formulation and testing of hypotheses, either by experiment or the use of internal models. Rules allow the application of methods known to work from past experience, by recognizing their applicability based on the situation or problem encountered. Rules form the basis for current expert systems, and represent the material gained by knowledge elicitation techniques. The difficulty of working within a knowledge-based performance domain presents a serious obstacle to the activities of the interactive system designer, and underscores the value of rules and procedures in the design task in general.

5. RABBIT: an example of a futuristic system

RABBIT (Williams 1984) provides an example of a current system that achieves some of its power from the sharing of knowledge in the form of explicit models. The main emphasis is on just the process of explanation, refinement and conceptual consistency. RABBIT is a database retrieval system that must show its view of the world to the user in such a way that the user can formulate retrieval questions in acceptable, albeit crude, terms. Then a process of successive refinement occurs in which the user moulds the crude physical form of a question to correspond to the intent of the particular retrieval, but framed in RABBIT’s terms. As the refinement proceeds, the effects it has on what will be retrieved are made apparent. Also, by noticing the physical details of most interest to the user, RABBIT is able to adapt the information presented to fit this interest. In setting up the query formulation this way, Williams is faithful to the design-interpretation model of communication promoted by Thomas and Carroll (1981) and, at the same time, ensures that the means and ends are continually in view and under control, linked by the functional
mechanisms of the retrieval system. When RABBIT fails it does so exactly at the point where the user (e.g. a wine novice) is unable to establish a connection between the means available (the wine expert terms and concepts understood by RABBIT) and the intent of the query (e.g. to choose a wine for dinner). Here, an expansion of RABBIT’s world view, and some user modelling along the lines of Rich’s *Grundy* system (Rich 1983), would presumably solve the problem. The point is that effective action is based on shared models that either the human or the computer may update for purposes of communication, improved performance, or whatever. These processes also take place in strict sub-servience to human goals and intentions.

Williams is quick to point out that the interface design for RABBIT is highly specific to information browsing, exploration and retrieval and has not been extended to interfaces for other tasks such as word processing, or computer programming. At first sight, it might appear that the ideas do not easily transfer. Indeed, Moran’s view that designing the interface is tantamount to designing the user’s model [of the system, and hence designing the system itself] (Moran 1981), could suggest an opposite view, namely that every system has unique interface requirements. In a sense, this is quite true, but certain generic ideas do transfer. The best word processors depend critically on the idea that ‘What you see is what you get’ (WYSIWYG), and thus obey the same design-interpretation model of communication between operator and system that RABBIT follows. It is fairly easy to imagine a programming paradigm along the same lines (see subsection 4.2). In fact there is a strong flavour of the design-interpretation model in the communication/design of algorithms using interpretive languages, which goes quite a way towards explaining their attractiveness.

Although more conventional programming systems do not provide quite the same facility for working with the solution model at several levels at once (especially given the relatively crude debugging aids available), the essence of programming is successive refinement, and iteration towards completeness and correctness. Despite the claim that top-down design is an orderly process that results in a correct solution when complete, and despite the goal of automatic programming to produce correct programs from provably correct specifications, somewhere along the line the purpose of the human has to be matched to the means for achieving that purpose. Even mathematicians admit that the discovery/creation of a proof is not the orderly process suggested by its explication. At the point where the human skills are applied, the same iteration between models and levels of representation must take place.
There is plenty of scope for development and experiment to learn more about this process, and possible related algorithms. It is worth emphasizing the implications of a previous point here. To the extent that task completion in an interactive system depends on the construction, modification and reconciliation of models at different levels, future computer systems must make these models explicit, examinable and manipulable by the users, in their own terms. Communication with the system about the models, their relationships, and their form must follow the natural effective design-interpretation schema. And, furthermore, changes made at one level must immediately be reflected at other levels, in appropriate terms. Research on solving the associated problems will be a key factor in the proper design of future computer systems.

The central issue in designing interfaces for future computers will be the formation, representation, manipulation and use of models. This kind of model-based activity is central to human perceptual, communicative and problem-solving processes (e.g. Gentner and Stevens 1983). Rissland (1984) raises essentially the same point when she talks about the importance of sources of knowledge. Like knowledge, models can exist at various levels of abstraction, and may be built up in a hierarchy. Models represent sources of knowledge for planning, testing and evaluating in knowledge-based performance, or for customizing the interaction to both user and task, setting up representations, domain knowledge, and tools. One kind of model represents the kind of generative core knowledge that allows a user to avoid rote memorization of procedures in dealing with complex systems, reconstructing them instead, and even generating new ones (Halasz and Moran 1983). Other kinds of models represent information about the user, the user’s understanding of the system, the task, and methods of interaction. Some of the models must exist within the system, and some within the user, but, in a very real sense, all these models must be shared. Even inaccessible models must be shared in some form.

A metaphor is a partial model, built to represent some aspect of reality, that is useful in understanding some other aspect of reality, in this case, an interactive computer system. The topic is considered by Carroll and Thomas (1982) and Thomas and Carroll (1981). Interaction will be more effective when the models and metaphors are closer to the reality they mirror, unless the task is strictly routine (i.e. rule-based) (Halasz and Moran 1983). The system can actively assist the user in forming correct models of itself or suggest appropriate metaphors. The user can contribute to the system’s models of the user and the current task (in RABBIT, this is the role of
Part of the role of interaction is the updating and correction of the relevant models, which involves both tutoring and knowledge elicitation, but a great deal of knowledge can also be built into the system. The success and power of SOPHIE (Brown 1975) as an instructional system depended in part on the excellence of its modeling and reasoning, and in part on its ability to communicate using language in a natural and robust manner, both depending on a great deal of built-in knowledge (circuit simulation, natural language parsing, spelling correction, ... ). The modification of the circuit model to represent faults, and the reconciliation of the fault model with observation, gives an early example of the kind of knowledge-based interaction that will come to dominate future computer systems, as more routine tasks are automated almost completely (that is, automated apart from the potential for human intervention). Models are, in a fundamental sense, the ultimate development of object-oriented programming, since they encapsulate data, and the procedures for operating on that data, in an absolute, literally ‘real’, sense. In rule-based performance, the models will be static, and will be available only in terms of their inputs and outputs. In knowledge-based performance, dialogue must take place at a meta-level, and the objects become accessible for internal modification. Models that exist in the human, and are inaccessible for various reasons discussed earlier, will constitute objects not available for internal modification from the computer’s point of view. Some computer objects will equally be unmodifiable by the users, perhaps depending on the role and expertise of the user.

One interesting aspect of the role of models, hypotheses, and levels of abstraction is noted by Norman (1984). He is concerned that future interfaces should move away from the level of details (physical models) towards the intentional global levels. He relates the story of a man going to open his car door:

‘X leaves work and goes to his car in the parking lot. X inserts his key in the door, but the door will not open. X tries the key a second time; it still doesn’t work. Puzzled, X reverses the key, then examines all the keys on the key ring to see if the correct key is being used. X then tries once more, walks around to the other door of the car to try yet again. In walking around, X notes that this is the incorrect car. X then goes to his own car and unlocks the door without difficulty.’

He reports having a collection of stories similar to this, revealing that even though people know their own intentions, they seem to work bottom-up, tackling the problem at the
lowest level, and only reluctantly and slowly moving to the higher levels of action and intention. There is a role here for an interactive system to prod users out of inappropriate levels, and away from incorrect hypotheses. Suppose the door could have said ‘That key is for a different car’.

It has been established that one common failure mode in human problem solving is failure to abandon an initial hypothesis. In one study (Wason 1971), students were asked to determine the rule underlyng the generation of a number sequence, given the beginning of the sequence. If the rule was incorrect, further numbers in the sequence were given, refuting the initial hypothesis, and providing more data. Many of the students simply reformulated the original, incorrect hypothesis, perpetuating their inability to solve the problem. This behaviour is seen in Norman’s example, except that, being in a richer environment, the subject is eventually forced to abandon successive hypotheses until the solution is literally forced on his attention. T. F. M. Stewart has called this the ‘setbreaking’ problem.

These examples also capture another essential of interaction with future computers, namely the importance of redundant, multimodal exchange of information. The issue is raised explicitly in a chapter entitled ‘Future Interfaces’ in Bolt’s book about the work of the Architecture Machine Group (Bolt 1984). Noting that related information in a communication channel may be redundant or supplementary in character, and that this form of communication was invented by nature, he points out the advantages of being able to speak, point and look, all at the same time (Dataland allows all these modes to be sensed). Supplementary information allows such possibilities as the resolution of pronouns by pointing (making speech more economical and natural). Redundant information allows correct identification of intent from information that, taken piecemeal, is ambiguous because of imperfections of various kinds. Bolt also emphasizes the importance of integration of sources of information. The usefulness of the whole is greater than the sum of its parts.

The whole concept of the AMG’s Dataland is futuristic, but in summarizing the future interfaces chapter, Bolt picks out two other main points as especially relevant. One is the use of circumstantial cues, particularly in retrieval, and the other is the use of eye tracking to help in modelling the user’s interests and intentions.

We tend to remember circumstances, even when contents are forgotten, and even though the circumstances may have little formal connection with the desired fact or action. Thus we remember information from books, reports and newspapers partly

6.3. Redundant, multimodal communication: pointing, looking, and situational cues
on the basis of where and when we obtained it as well as whereabouts within the source the information was encountered. Such information provides hooks to access our associative memory, and explains why examinations seem easier if they are taken in the lecture room, rather than some new place. Far from preserving this kind of information, current computer systems even suppress what little may exist. Thus text, presented on screens, is likely to change in format, depending on the terminal, or the trivial modifications made since a previous visit, whilst failing to give anyone document distinct visual character. A future computer system can keep the circumstantial record of a user’s activities and preserve the distinct idiosyncratic form of documents even in electronic form, using them to assist in future interactions. Printed material may, in future, only be viewable in bit-mapped run-off form, which could also help in copyright protection (Benest and Jones 1982). Cheaper approaches to structured document viewing that tie in with document preparation are also possible (Witten and Bramwell 1985).

Eyes, Bolt notes, are especially revealing. They form a highly mobile pointer, revealing interest and focus of attention, as a supplement to dialogue. A great deal can be communicated by a changing point of regard, both intentionally and unintentionally. A child learns the names of things by hearing the names and noticing what the namer is looking at. Bolt distinguishes three kinds of looking: spontaneous; task-relevant; and changing orientation of thought. In addition, there are pupil size effects that relate to degree of interest as well as stage of task completion. Thus, although there is clearly a need for more research, in principle it is possible to determine what a user wishes to know about, how interested the user is, and how the user’s mental tasks are progressing, especially when coupled with other cues like voice and gesture. This, and other multimodal input can be used to form and update appropriate models related to the overall management of the interface.

The control and management of human-computer interaction in future computer systems will depend on the success of research currently in progress. An excellent review of the state of the art appears in Pfaff (1985). Present attention is focussed on the functional divisions within the overall User Interface Management System (UIMS), on the location of control (in the application or in the UIMS), and on the nature of the method used for formal specification of the dialogues that are the object of the UIMS. The UIMS, which mediates between a user and an application, is intended to provide a framework for the construction and management of user interfaces that
cuts out repeated hand coding of common parts of human-computer interfaces, allows complexity management, and provides uniformity, consistency, and other desirable properties in the resulting interface by the constraints and facilities it embodies. Given the formal specification, it allows certain kinds of error and interaction performance to be verified. And, perhaps as important as any other advantage, a properly constructed UIMS allows interactive systems to be prototyped very rapidly, with user involvement and feedback. This is so important in practical applications that one should really talk about User Interface Prototyping and Management Systems (UIPMSs).

The argument about location of control is reminiscent of the arguments about graphics packages versus graphics languages. In a graphics package, or an internal control UIMS system, the interaction is controlled from within the application, with the package encapsulating the appropriate graphical or interaction techniques for use by the application. In the graphics language, or external control UIMS, the system (graphics system or UIMS) is in control, and calls applications resources just like any other resources. Some authors (Hayes, Szekely, and Lerner 1985) suggest the use of a mixed control UIMS, to try and obtain the advantages of both, whilst avoiding their disadvantages. It seems likely that none of these solutions is entirely satisfactory. A better approach is likely to place control at a higher level than either the application or interaction resources, forming a task management level, as suggested in subsection 2.5. Such a system would represent an operating system component, avoiding the question of whether the application or the interface had control. The form of such a solution is still a subject for research, but will certainly involve a more methodical approach to the instantiation of applications resources. Indeed, an important aspect of future computer systems will be the determination of adequate applications primitives, as well as interaction primitives, file management primitives, and the like.

It is not possible to summarize all the arguments and problems associated with various aspects of UIMSs within the scope of this paper. However, various expert system components within a UIMS will be needed to represent at least

- the interaction desired (a script for interaction (e.g. Hill and Irving 1984),
- the task (intentions, goals),
- appropriate problem solution techniques (applications primitives),
- the user (e.g. Rich 1983),
• appropriate interaction techniques (e.g. Foley, Wallace, and Chan 1984),
• activity metacommentary (modelling techniques),
• and the system itself (intelligent help and file management, etc.),

to provide the models, or sources of knowledge, demanded by Rissland’s exposition of the intelligent user interface (Rissland 1984). It seems clear that the UIPMS component of future interactive systems will be a sophisticated expert system, well integrated into the basic operating software of any future computer. The UIPMS will require an interface of its own, which would provide a uniform basis for interaction for all users. Like software for other computer methods, the UIPMS would be much easier to design and implement if it were already available to assist in the task, but a bootstrapping approach will have to suffice, given the framework. The ultimate development of the idea would conceivably eradicate and distinction between programmer and non-programmer by making problem solving and/or the definition of problem solving methods effective, productive and fun for anyone with a problem to solve and access to a computer. That was certainly Smith’s ideal. But then, that is what the inventors of FORTRAN hoped.

7. In conclusion

This paper has examined the drive towards better humancomputer interfaces in the context of the needs of future computer systems, has raised issues that seem important in this quest, and has highlighted areas where work has started on some of the deeper problems that must be solved to provide appropriate interaction with future computer systems. The real difference between the creations of the AMG (Dataland), and Arthur C. Clarke (HAL), from a conceptual point of view seems relatively small. Both represent views of the future, albeit with only a sketchy supporting framework. Superficially, apart from the quality of the various modes of interaction, and an ability to lip read, HAL offered little as fiction that is not already available as fact for interaction with Dataland. However, Dataland is still just a sophisticated information retrieval system, lacking either problem-solving ability of its own, or the ability to guide a human in such activities. The information modalities, and means of interacting with them are clearly very sophisticated, and have advanced our understanding of data forms and interaction techniques, including the integration of different modalities. But it is just a start.

The central issue for interaction with future computers will be the formation, representation, communication, manipulation, and use of models embodying all kinds of useful
knowledge in an accessible form, so that it can be applied to make interfaces truly supportive without entailing the abdication of responsibility on the part of the human user. The computer will provide the intelligent computational Indians for the human Chiefs. This situation will demand both more and less from the human interacting with the computer. All kinds of knowledge on facts and procedures will become readily available, to the extent that it can be formalized, and access to computers will move closer to the model espoused by Foley and Wallace—that of natural conversation, whatever the medium. But the human user will have to understand much wider problems at a higher level. To the extent that users have expertise and solve problems, they will add to the store of knowledge in the computer and explore its limitations. It seems likely that the kind of interaction envisaged will promote the sharing of knowledge in both directions, so that education will become an ongoing, on-line experience as computer users pursue their careers. Again this will both help, and demand more from, the human. However, the distinction between programmer and user will remain. But the programmer could very well be called a knowledge systems therapist, concerned with the nature of the world, the understanding of knowledge, and the care and development of the machine. In this context, perhaps Weizenbaum will be in danger of losing an important element from his line of argument, as we socialize our computers.

Acknowledgements

The author wishes to acknowledge with gratitude the financial support of the Natural Sciences and Engineering Research Council of Canada under grant A5261. The author would also like to thank those who read and commented on earlier drafts of this paper, especially John Andreae, Bruce Conrad, Richard Esau, Dan Freedman, Brian Gaines, Saul Greenberg, Harold Thimbleby, Ian Witten, and two anonymous referees. Any remaining problems are, of course, the author’s responsibility as he made the final decisions.

References


MORGAN C. (1983). 'An interview with Wayne Rosing, Bruce Daniels and Larry Teslar,' Byte 8(2) (February) 90.


SMITH M. J. (1975). When I say NO, I feel guilty (Bantam Books, New York, 324 pp.).
Many computer centres in North America and some elsewhere, including university departments of computer science, are connected by several networks that allow all kinds of material, including personal mail, to be widely transmitted and received. The following material arrived over one of these networks. The originator's network addresses (CSNet, ARPAnet and uucp) are all appended as the last three lines of the message. The main text represents computer prompts ('%') and replies, with user input in italic. The script follows the general form of a popular adventure game Zork, but instead of travelling underground, fighting, picking up bottles, axes, and the like, and throwing them at dwarfs or collecting water in them, and finding treasure, the objects and characters and plot are within Unix. It is not unlike the movie Tron, in which a real character enters the conceptual world inside a computer.

> From: PHIL REED  
To: USER FRIENDLIES  
Subject: AN IDEA  
Date: Wed. 12/09/81 10:56  

Has anyone thought of grafting the pseudo-English parser of Zork or other dungeon games onto a shell? Not only would it be a cheap way to get the more natural syntax some people say they want, it could provide a very amusing interface, and add glamour and excitement to ordinary work:

%go to bin  
you are in Jon/bin. there are many files here.  
%look  
you are in Jon/bin. the directory contains:  
date date.c readtape scramble  
the only exit is up.  
%use date to create tmp  
done!  
%take tmp  
taken.  
%open tmp  
you open tmp to reveal:  
Tue, November 24, 1981 (2:00pm EST)  
%take date.c  
taken.  
%look  
you are in Jon/bin. the directory contains:  
date readtape scramble  
the only exit is up.  
%inventory  
you are carrying: date.c tmp  
%goto src  
you are in Jon/src, there are many files here.  
%drop date.c  
dropped.
%throw tmp at printer
the printer catches your file, turns it into paper, and leaves it in the basement.
%look
you are in Jon/src. the directory contains: date.c scramble.c readtape.c
there are exits marked ‘zshell’ and ‘secret’ as well as the path leading up.
%xyzzy
you are back in your home directory. there are many files here.
%run foo
the foo dumps core
Oh dear! you seem to have gotten yourself killed.
%attack core with sdb
...

End Inserted Text.

—

Off the Wall of Gene Spafford
The Clouds Project, School of ICS, Georgia Tech, Atlanta GA 30332
CSNet: Spaf@GATech
ARPA: SpafGATech@CSNet-Relay
uucp: ... ![akgua,allegra,r1gvax,sbl,unmvax,ulysses,utsally]!gatech!spaf

Appendix B: Principles for human-computer interface design. 1986

In what follows, numbered bold items indicate selected principles for design, items (1) and (2) being of pre-emptive importance. Under each numbered category, elaboration of the principles occurs, and curly-bracketted entries { ... }- indicate, in note form, the relevant human and/or machine consideration(s) involved.

Like many categorizations, this one is probably somewhat arbitrary. Other divisions are undoubtedly possible, and the inevitable overlap and uncertainty between the categories makes them less clear-cut than one would like. However, the selection and structuring does attempt another small step towards the goals of formalizing the user interface design process, and giving more detailed guidance on the ‘whats’ and ‘whys’ of the process. It does not attempt to provide the kind of formal framework for specification aimed at by Moran (1981), but it does provide a structured guide concerning what goaloriented content should be fitted within such a framework. The categorization now follows.

(1) Know the user.

The designer should be intimately acquainted with the user’s needs, the user’s frame of reference and experience, and the conditions under which the task(s) will be performed. The less this is satisfied, the more likely it is that the interface design will be deficient in meeting those needs, and in providing a natural, comfortable, non-
intrusive tool. Knowing the user also includes knowing about the limitations, strengths, skills, and characteristics of humans in general.

Investigate the characteristics of the user population directly (not second-hand: stereotypes, percentile measures, conceptual framework ... ) and design accordingly. The designer must remember that he or she is not necessarily a good example of a typical user, even if there is considerable overlap in terms of experience and task characteristics with some users. Self-assessment of interface components can often short-circuit the need for extensive human factors experiments, and guide design, but can be very dangerous if carried too far. Presumably every system is ‘friendly’ to its designer.

Thus the designer must appreciate that humans vary greatly in their physical and mental characteristics, being especially aware that different does not usually mean inferior. People have valid preferences, that often reflect the very experience that the designer should be exploiting. There are a number of important aspects of the human operating characteristic, including the fact that mistakes are inevitable, whilst fatigue, boredom, panic, and frustration cannot rationally be condemned, but only avoided by careful task and interface design. Humans also have strengths and abilities denied to machines—that is why they are designed into the system in the first place—and the designer should build on these qualities, and design to overcome the less convenient human characteristics. Indeed, learning to use neutral terms to discuss such problems, rather than using terms such as ‘weakness’, ‘failure’, ‘operator error’, ‘impatience’, and the like is probably half the battle in meeting these aspects of the design goals.

(2) Design the tools the user needs, fit for the user’s tasks.

If a system is designed that does not provide the specific task-oriented components needed by the user, in suitable form, then it is deficient in a very fundamental sense. Application of the first principle results, amongst other things, in a detailed statement of the specific tasks that the user must perform. The next step in the design process is to design the tools to meet each particular user need in the specific task context. The tools should be consistently integrated into an overall system that fits the user’s conception of the task and task environment. Perhaps the best approach to ensure this is done effectively is to start writing the user manual at the earliest possible stage in the design process. If possible, a dialogue prototyping aid should be used to help formulate the design realistically, to allow the effect of design decisions to become obvious before they become cast in stone, and to give the user direct experience (promoting user involvement in the design process).
A reasonable outline for such a manual would be as follows.

- **Summary.** Briefly summarizes the content of the document.
- **Introduction.** Gives background to the application. What is the general area into which the task falls. Who are the users and what are their overall needs. Who was consulted. Any special circumstances or difficulties ....
- **Purpose of the system.** The specific purpose of this interactive dialogue—what job does it do for the user. Be brief. Details can go in the section on capabilities or transaction details.
- **System overview and rationale.** An outline view of how the system is organized, and why it is organized this way. A diagram showing the main blocks, paths and relations may help here.
- **System capabilities.** A list of the specific capabilities of the system, rather than just a statement of the overall function as stated in the ‘Purpose’ section.
- **Transaction details.** This, and the next three sections, are the most detailed part of the user manual. An introductory subsection should explain overall screen formats and the like; and then the actual screen formats, error handling, default entries, form of feedback, any other keystroke saving facilities (e.g. menu short-cuts), and so on should be outlined for each transaction. Common features (such as editing modes) go in the introductory subsection.
- **Help facilities available.** How help is organized and how to access it.
- **Facilities for audit and gripes.** (How they are organized and how to access them.)
- **Sample dialogues.** A few representative samples of typical dialogues, using reasonably exact representations. The main point is to give an idea of the system in use, as opposed to the previous section which tends to give a picture of the parts, without relating them. The overview diagram from the overview section may be a useful aid in explaining how dialogue sequences work out.
- **Critical review of system with a note of ‘next-release’ improvements.** Stand back and try to point out any problems with the system. This section provides a guide for anyone who might have to produce the next release.
- **Acknowledgements.**
- **References.**
- **Appendices.** The most likely items here are forms associated with some existing system.

The remaining principles, which serve as categories within which design rules can be successively refined, really follow from the two major principles above, and cannot be applied in a vacuum. The designer must have a goal, a functional context, and an understanding
of the available materials in order to apply principles and instantiate design rules in an effective and integrated manner.

(3) Make the system easy to learn and remember. (See, also, (8) below.)

Keep the system as simple as possible whilst still meeting other criteria. Ask what can safely be excluded, not what might be put in.

{Short-term memory is limited-magic number 7 ± 2.}
{Too much detail confuses, and slows human operations.}

Be both consistent and uniform in the style and details of the dialogue. As Gaines has succinctly stated (Gaines and Shaw 1983):

‘All terminology and operational procedures should be uniformly available and consistently applied throughout all system activities.’

Use familiar terms and familiar concepts.

{Humans learn new skills in terms of past experience.}
{Human memory is associative.}
{Negative transfer occurs between incompatible skills.}

Use mnemonic coding for all things symbolized (icons can also be mnemonic). (A mnemonic symbol is an easily remembered symbol having strong association to the item symbolized.)

{Associative character of human memory}

Facilitate the formation and use of models: the user should be taught an adequate model of the system, and the system should acquire and use information about the user and his or her goals. The user’s mental model is central to this. It constrains the design in the first instance, and serves as a major link with the user in learning, using and improving the dialogue.

{Humans need structure to combat complexity.}
{Humans work best in terms of their own conceptual models.}

Maximize continuity in all aspects of the interaction; visual, tactile, contextual, command language use, layout, stereotypes developed, ... (cf Foley and Wallace 1974).

{Physically and perceptually obvious}
{Models, and transfer of training}

Provide excellent help facilities. Help should be specific to the context where it is needed; the user should not have to work out how to access information on something he or she doesn’t understand. It should be invoked by some obvious action, probably even automatically in certain cases. Initial help (especially
Interacting with future computers

if automatic) should be succinct. Repetitive requests should increasingly detailed help (Gaines and Shaw 1983: *Query-in-Depth*). The user may quite possibly need help with the help facility itself, to avoid interactive deadlock, even if only by an expandable index richly laced with synonyms (a *Help-Help* facility).

{Well designed help is a powerful learning aid.}
{Human memory is fallible.}
{Don’t give a person a job if the job can be defined so a machine is better.}
{Even experts cannot always remember everything.}

Optimize learning/skill-acquisition; provide specific aids to learning. Note that learning requires support and takes time to happen—it is a biological process.

{Humans are adaptable, but also need to learn to use the system.}

Understand what the human users are trying to do in their terms and try to help them do it.

{Humans work best in terms of their own conceptual models.}

Present system functions and facilities in a form appropriate to the user’s skills and background. Remember, too, that even experts have to learn, may forget, and also evolve in terms of the use they make of any system, so that, even for experts, easy learning must be provided. In any case, experts can get quite frustrated when they are not on their home system. Indeed, this may explain why ‘experts’ tend to be so partisan about the operating systems and languages they use. Furthermore, the system must cope with staff turnover.

(4) **Deal with errors in a positive and helpful manner.** (See, also, (8) below.)

Avoid errors if at all possible, without restricting the user. Thus a cursor-selected menu prevents unknown commands, but may restrict an expert.

{Mistakes are an inevitable accompaniment to human performance; the more errors are possible, the more will be made.}

Provide assertive explicit error messages that identify the cause of problems and set users on the right track to correct them (cf M. J. Smith, 1975—an ‘assertive’ message carries no emotional overtones—is not patronising, obsequious, or rude).

{Humans make errors.}
{Humans need to maintain a good self image.}
{Humans do not perform well if they do not feel in control.}
Provide ‘reasonableness checks’ on input data, and correct ‘obvious’ errors automatically, normally notifying the user (unless the user explicitly chooses to disable notification).

{Humans make errors.}
{Humans should not have to do things the machine can do well.}

Don’t make the user feel stupid or put down by a literal, unchecked interpretation of user input when the resulting action would be inappropriate, or would have serious consequences. This clearly requires some model of task and goals.

{Maintain a good self-image for the user.}
{The machine should be a ‘good servant’}

(5) **Consider, protect against, and, if possible, avoid both human and machine failure modes.**

Make the system ‘bullet-proof’ so that it cannot be crashed, and control exit from the dialogue to other host facilities.

{Humans are deranged by unexpected action.}
{The computer should be a ‘good servant’}

Allow forestalling of prompts. That is, first allow type-ahead. Then, if the user has anticipated a prompt, and given the response already, drop the prompt. This can automatically provide for shortcutting menu hierarchy traversal and is of especial importance when speech output is used.

{Humans do not function well when subjected to delays.}

Make simple repetitive tasks the responsibility of the machine.

{The machine is tireless, good at repetition, and direct memory tasks; humans are not.}

Aim for optimum stress.

{Humans become deranged if bored or overloaded.}

Provide back-up for machine components in humancomputer systems, especially critical real-time systems (of course, human back-up may also be needed).

{Machines tend to fail suddenly and completely.}
{Machine failure will almost certainly overload the human capacity.}
{Humans may panic when subjected to unexpected emergencies.}

Be forgiving and flexible. Support not punish.

{Humans do not handle rigid protocols well.}
{Humans become deranged by perceived lack of cooperation.}
Provide variety to motivate and interest the user (though this conflicts with consistency, and is secondary; initiative on the part of the computer frightens some and stimulates others).
   {Humans enjoy variety. It motivates them if they still feel in control.}

(6) **Provide good feedback on what the system is doing, and how this fits in with the overall structure of the system.**

Provide clear feedback to the user, especially for errors, but always so that the user is sure of what is going on.
   {People perform better, learn better, feel more in control, and develop a better model of the system with feedback, which increases motivation, builds confidence, and promotes accuracy.}

(7) **Structure the interaction, the information presented or requested, and any data used, to help the user cope with its complexity.**

Break up any information for recall or choice into familiar groupings:
- familiar organizations, e.g. mimic diagrams, relate closely to a user’s reality;
- exploit chunking and association, especially in choosing mnemonics;
- pictures are worth many words (but may require commentary);
- layout, fonts, colour, ...
- hierarchical, or other appropriate organizational structure.
   {Short-term memory characteristics.}
   {Humans find unstructured detail difficult to manage.}

Provide structural cues to combat complexity
   {Don’t leave the human to work out something if the machine can help.}

(8) **Convey a real feeling of control to the user, even when it is the computer that is asking the questions.**

Make it easy to ‘escape’ and to correct errors. Confirm actions having major consequences if recovery is not practical. People often don’t appreciate the consequences of their action until too late, so that an *UNDO* command is a valuable approach to this problem. A system that is uniform, consistent, and easy to learn and remember promotes a feeling of control (see (3) above).
   {Humans do not function well if they do not feel in control.}
   {Humans do not feel in control if they find it impossible to escape from a course of action, or reverse the effects of a mistake.}
Humans make errors, it is part of their operational nature and designers must design on that basis and allow for the inevitable. *Note: this actually reduces errors because of reduced anxiety, apart from the direct effect on productivity.*

Humans will learn more quickly and effectively if it is easy to experiment without horrible consequences.

Make provision for the user to tailor the interaction to his or her needs

The user must feel in control and be in control. Individual differences—people vary.

Minimize the activity needed to initiate actions—but don’t penalize verbosity.

Humans do not feel in control if control requires great effort. Individual differences—people vary.

Don’t have the machine do unexpected or unreasonable things.

Humans are good at initiative and creativity; if the machine competes in this, panic may ensue, the image of the machine as a good servant is destroyed, the user’s model is disrupted, and the user no longer feels in control.

(9) **Consider carefully the division of labour in allocating tasks between human and computer.**

Capitalized on the strengths, and avoid the less convenient characteristics, of both humans and machines. For example, divide tasks and present information in a way that allows the human to exercise integrative skills effectively (e.g. by pictures that reveal matters of importance, rather than tables of numbers).

This is the prime reason for having the human acting in cooperation with the system to start with!

Human abilities and machine abilities are complementary.

Minimize keystrokes within the constraints of familiar mnemonic, redundant coding for all symbolization. For input, function keys can be used whilst allowing multicharacter symbols. Voice buttons that work provide an ideal access to functions, except they are not self-documenting. Providing sensible defaults for input (including commands) wherever practical is another excellent measure that can be taken.

Physically obvious; don’t have the human doing unnecessary work.

Function keys aid memory, as well as reducing keystrokes and promoting association.

Less chance for errors.
Provide for the machine to do all arithmetic—either automatically, or by a calculator function.
{Humans are slow and inaccurate at arithmetic.}
{Use of a real calculator breaks continuity.}

Use documents, where they perform best for people.
{Humans scan documents easily, and find them more convenient than screens for some purposes such as browsing and searching for answers to ill-defined requests, or reading in the bath.}

**10) Take into account human performance limits.**

Present information legibly.
{Human performance limits.}

Don’t overload the human communication capacity.
{Not only straight information loss, but also the multiplicative effect of confusion.}

Allow enough time for the human to function.
{Humans have limits to their information processing capacity.}

Consider carefully the arrangement of the physical workplace.
{Humans have limits to their physical reach, direction of gaze, ability to see when subject to glare or reflections.}

**11) Ensure that the user is psychologically comfortable.**

Provide adequate rest periods for human operators.
{Humans require rest and recreation to maintain motivation and performance.}

Make the system aesthetically pleasing.
{Humans perform better when they feel someone takes an interest in their working conditions.}

Don’t ‘put down’ the user.
{Humans need a good self-image to operate properly.}

**12) Ensure that the user is physically comfortable.**

Arrange a comfortable workstation.
{Discomfort leads to fatigue and distraction, hence errors and lowered productivity.}

**13) Design is an ongoing process. Provide facilities to monitor overall system activity.**

Provide an audit trail for tracking down problems, as well as for security. Log system activities (this requires careful selection and processing of important, necessary material, plus the discard
of all else, or there will be an unreasonable amount of unstructured data; it is not reasonable to expect that this kind of log is a substitute for carefully designed experimental data collection intended to support the testing of hypotheses in evaluation trials; it can only act as a focussing mechanism). Make explicit provision for user complaints and suggestions (a ‘gripe’ facility) that is easy to use, as a dialogue excursion. You need the information.

{Don't ask the human to do alone what the machine can help with: reduced effort increases response likelihood.}

{Humans perform better when they feel someone takes an interest in their working conditions.}