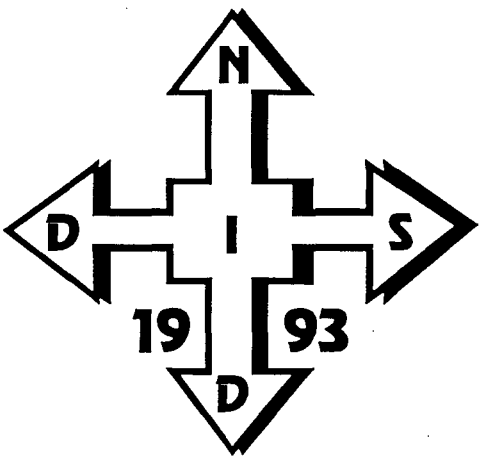
UNIVERSITY OF WOLVERHAMPTON

BCS
THE SOCIETY OF INFORMATION SYSTEMS ENGINEERING



NEW DIRECTIONS IN SOFTWARE DEVELOPMENT 1993

HCI: MAKING SOFTWARE USABLE

MARCH 1993

UNIVERSITY OF WOLVERHAMPTON
SCHOOL OF COMPUTING
AND INFORMATION TECHNOLOGY

PROCEEDINGS OF THE

SEMINAR SERIES

ON

NEW DIRECTIONS

IN

SOFTWARE DEVELOPMENT

HCI: Making Software Usable

March 1993

EDITED BY: M.A. Garvey
SCHOOL OF COMPUTING AND INFORMATION TECHNOLOGY
UNIVERSITY OF WOLVERHAMPTON

SPONSORED BY THE INTERNATIONAL JOURNAL:

INFORMATION AND SOFTWARE TECHNOLOGY

CONTENTS

HCI: Making Software Usable

In the good old days people were quite content with a keyboard and screen, today people's aspirations have changed and they are contemptuous of a computer without at least a mouse attached to it. Given this new level of sophistication, are we any better off in terms of usability, or have we just been give a plethora of extra gadgetry to further confuse us?

Outside the computing profession, people have high expectations of what a computer can do. Some would like the ability to *talk* to the computer. Star Trek fans will recall the scene where Scotty tries to talk to one of our *ancient* Macintosh's via a mouse. In the future we may be able to communicate with our computer verbally, but the 23rd Century is a long way off. At the moment we have to contend ourselves with the ubiquitous WIMP.

What is the definition of good interface design? Applications developers may get carried away with the latest functions their software offers. All very well, but the poor user may end up with a screen crammed with an over abundance of popup menus, stylishly designed in the latest magenta on cyan colour scheme, not forgetting the red flashing *help* messages. The application may be high-tech, but is it usable and aesthetically pleasing?

Those of you who have programmed a video appreciate the importance of good interface design. In order for a computer application to be accepted, it too needs to be designed so that it is both easy to use and not garish to the eye.

User Friendliness, Man Machine Interfaces and User Interfaces are all terms associated with HCI. Whatever the terminology used, the goals should remain similar in that it should aim to improve the effectiveness, efficiency, safety and usability of any computer system. HCI is an important part of systems design, it is about assessing the task in hand, the user and environmental factors, so that the design will result in a system that is effective for the situation.

Usability entails making the system easy to use and learn. This means stopping the system becoming over complex. The ways of achieving this can range from reducing the number of keystrokes at the low level, to restructuring the system to model the task in hand.

Practitioners of HCI should aim to understand how people operate, develop tools to aid designers address user needs and finally achieve suitable and effective interaction

Today's talks address some of these issues, starting off by looking at Software Usability. Once developed, a system may cover all the functionality requested by the users, yet can still be rejected because it requires obscure keystrokes to do anything. Andrew Lees explains how to avoid such situations.

Jonathan Crellin discusses PROTEUS, an interface evaluation technique. He addresses the current difficulties in interface evaluation and shows how PROTEUS offers a solution. Many of these problems are due to the fact that the designer of a system may have a different outlook on interface design compared to the user.

Ernest Edmonds looks at issues of separability. He explores how the *User Interface* can be identified as a separate component. The seminar reviews the history of separable user interfaces, looking also at suitable architectures, going on to address the practical problems that have arisen from this approach.

Alan Dix looks at the interplay of formal methods and HCI. Formal methods can require a high level of mathematical understanding, however this seminar will present two methods which have their roots in formal analysis, but which do not require great formal expertise.

Finally Phil Puttick finishes off by addressing the five most popular methods that can be used to confuse your users, giving us an insight in how to avoid them, gained from IBM's experience of system design.

Mary Garvey
School of Computing & IT
University of Wolverhampton

The emergence of the separable user interface

Ernest Edmonds
LUTCHI Research Centre, Loughborough University of Technology, UK

Abstract

The meaning of the term "user interface" is briefly reviewed and the concept of this as an identifiable component in a system is explored. The history of the development of separable user interfaces is reviewed, together with the growth of interest in user interface software systems and in techniques for user interface specification. Separable user interface architectures and practical problems that have arisen in the use of separable user interfaces are discussed.

Introduction

The user interface to a computer system goes under a number of names including the human-computer interface, the computer-human interface, the front end and the man machine interface. Whilst "front-end" tends to be used in the restricted domain of so called "intelligent interfaces" and "man-machine interface" is sometimes used to refer to systems that do not involve computers, for the purpose of this paper they can all be considered to be synonymous. Many papers and books on the user interface exist, including ones that give detailed advice on how to design and how to construct them. Nevertheless, there is still no universally agreed definition of the user interface. For example,

> "The human interface with computers is the physical, sensory and intellectual space that lies between computers and ourselves". (Bolt, 1984),
> "The user interface is that part of a system that the user comes in contact with physically, perceptually, or conceptually." (Moran, 1980)
> or
> "The functionality defines what the program can do, and the user interface defines how users tell the program what to do, and how the program tells users what it did." (Szekely, 1988).
>
> In fact many authors omit a definition altogether (e.g. Shneiderman, 1987).

In English, the normal use of the word "interface" is to name the relationship

between two entities, particularly where that relationship is well defined. Increasingly, however, the term "interface" has been applied to special devices that enable one of the objects to hold the given relationship with the other. Thus, for example, one might have to buy a special "interface" in order to enable one's computer to operate a particular printer. A similar tendency has occurred in the case of the user interface. The study of the relationship between the user and the computer is, now, most frequently known as the study of human-computer *interaction*. The user *interface* normally refers to the hardware and software that enables the computer to play its part in the interaction. It is widely assumed that it is possible to identify this interface as part of the system and, conversely, that it is possible to identify components that perform functions that should not be seen as part of the interface. It is this assumption that is the central concern of this paper.

Early work

Without doubt, the most important early work in this field was done as William Newman. This work built upon Sutherland (1965) in important respects but, nevertheless, represents the first significant steps towards the current concern for the object known as the user interface.

Newman (1968a) describes a virtual input device that enabled the user to enter numerical data on a continuous scale with the aid of a light pen or a similar device. Newman terms his logical device a "light handle" because it can be seen as a simulation of a handle that can be wound in order to increase or decrease the associated numerical value. The significance of this work was that it demonstrated the possibility of constructing what, today, might be known as an "interaction object": that is to say, a self-contained software module that uses the physical input/output devices available in order to enable the user to provide some particular kind of input to the machine. Whilst, in modern terms, this might be seen to be a very low level of separation, it represented an important conceptual step.

The most significant early work was, however, Newman's system for interactive graphical programming (Newman 1968b). A *network definition language* was introduced that partly defined interactive software in the form of state diagrams. A compiler for this language could generate run time code. This system was based upon a distinction between the *reaction handler*, with its associated tables, and the *procedure component*. The network definition language was compiled into the tables that the reaction handler used, as it were, to run the user interface. The procedure component was written in a conventional language that did not deal directly with any interaction with the user. The mechanism for linking the two aspects was the insertion of program blocks into the definition of each appropriate state of the state diagram. Thus, the network definition language defined the user interface together with its links to the functional part of the system: the procedure component.

Whilst Newman's Light Handle and his Reaction Handler both demonstrated the separation of user interface issues from functional ones, the distinction between them, in this respect, is very important. The separation offered by the Light Handle is that of an interaction object that can be evoked by the functional code but that looks after itself once active. The Reaction Handler is quite different in that it is in control of the system and evokes functional actions as required. It is, therefore, an example of a separate user interface. The paper did not, however, address the issue of the precise scope of concern of the procedure component. Indeed, at that time, this was perhaps not a major issue but more recently, as will be discussed below, this has become a significant point of debate.

Parnas (1969) also proposed the use of transition state diagrams for the representation of dialogues, although he only advocated them for the "top level" of user interface design and so, in his work, they did not lead directly to implementation. The links between the user interface and the functional components of the system were not associated with states, as in Newman's case, but with the arcs. The method was to label each arc with an input/output pair, representing the user input that would cause the arc to be traversed and the output that would result. Only simple examples are discussed and, as with Newman, no detailed discussion of the issues of user interface/functional component separation is given.

Denert (1977), in considering the proposition described above, introduced two more innovations. Firstly, he recognised the computational limitations of state diagrams, which represent finite state machines, and proposed the use of recursive state diagrams, which are much more powerful. Secondly, he dealt with the interconnection of the user interface and the functional component by introducing "task nodes" into the diagrams, which call the functional code. In some respects, this is similar to Parnas' scheme but it adds the possibility of different branching depending on the outcome of the functional operation. Thus we now have the concept of the functional code changing the course of the dialogue. It would seem that for the separation of the user interface from the functional code to be a reality for complex systems, both of Denert's innovations are required.

First generation user interface systems

During the 1970's a number of research workers followed William Newman's early example and constructed user interface software systems that, in various ways, provided for a separation between the user interface and the functional code. Stalling (1974) demonstrated a system for generating dialogue code that worked somewhat after the manner of a report generator program. Florentin (1977) elaborated upon this and noted the importance of the linking of the dialogue code to the rest of the system. However, neither author added significantly to the specific discussion of separability.

Maher and Bell (1977) specifically proposed a virtual machine as the key user

interface module. They illustrated their work with a user interface to a database and briefly discussed some problems that arose with the separation of concerns. In particular, they found that it was convenient to organise the database in a particular way for the benefit of the user interface, but that this conflicted with other requirements.

Black (1977) described another system which provided a separate "dialogue processor" which managed the user interface. A dialogue specification language was provided that compiled into a form directly usable by the dialogue processor. In this system, the dialogue processor produced a "target file" which contained the input required by the functional component. Clearly, this was a simple view by today's standards and so, again, problematic issues of separation were not addressed.

State diagrams are popular for the specification of user interfaces, but several other formalisms are also used. Green (1986) suggests that one of the earliest examples of a grammar-based user interface system was that reported by Edmonds and Guest (1977). It showed how a grammar-based language could be used to define an interface flexibly. It was used to construct a programming tutor and so was demonstrated as a feasible system. However, it is again true that the simplicity of the separation between user interface and functional system was such that the issues often seen now as the problems of separability were not discussed.

Ian Newman (1978) demonstrated how separation could be used to provide a variety of interfaces to a single system. Practical work, again, was done to prove the point but, as before, the particularly difficult issues of separability that are perceived today were not considered. Lafuente and Gries (1978) extended Pascal in order to provide what they called "dialogue frames", each of which could include a procedural block. Thus the structure of this user interface was somewhat similar to Newman's (1968b). The difference was that Lafuente and Gries provided an extension of a standard programming language, whereas Newman kept his interface code quite separate from the functional code.

User interface specification

Whilst the work on specifying user interfaces has not in general addressed the issue of separability directly, it is quite clear that the whole enterprise is founded upon the notion of separability. If it were not so, what would it be that one was specifying?

An important model for these specifications was Moran (1980) in which he proposed a view of the user interface that had a number of levels incorporated into three components: physical, communication and conceptual. A key point is that, in this view, only the conceptual component needs direct contact with the functional aspects of the system. In Moran's view the conceptual component contained two levels. The *task level* is concerned with

the structuring of the task domain, for the benefit of the user, and the *semantic level* is "built around a set of objects and manipulations of those objects. To the system these are data structures and procedures; to the user they are conceptual entities and conceptual operations on those entities." (Moran, 1980). The semantic level was seen as a way of representing the system's functional capability. This particular aspect of the structure of a user interface has proved quite fruitful in attempts to clarify the separation from the functional code. This paper, and its elaboration (Moran, 1981), proposed that the user interface could be specified in a well-defined set of levels. This work has been influential both in the study of user interface specification and, more specifically, in the study of architectures for separable user interfaces.

Mark Green (1981) reported a notation for the specification of user interfaces and presented it in the context of a traditional software development model. Whilst that model is no longer necessarily accepted (see for example Hekmatpour and Ince (1987) or an earlier questioning by Edmonds (1974)), the specification ideas are not invalidated. Green's notation includes the notion of atomic objects and operations that correspond to Moran's objects and manipulations. These represent the lowest level of the system of concern to the interface. Beyond that, is the concern of the functional code. This has been the standard view of separation. In his paper, however, Green introduces an important additional property at this level. He allows for the specification of *invariants*. These are relationships which must hold between the operators and objects. Whilst he only gives a simple example of an invariant, it is clear that it is entirely unrealistic to consider these objects as independent and that quite complex invariants may be required. The only question at issue is whether the invariants can and should be seen as the concern of the user interface.

Edmonds (1981) consolidated earlier work and put forward a specification notation that combined state diagrams with grammars, each having their appropriate role. This work continued the tradition begun by William Newman in that a compiler was available for the notation. The links between the user interface and the functional code were provided partly in the manner of Parnas, as described above, except that the input/output specifications on an arc could have a complex transformation relating them, so that specific inputs in the appropriate class could generate relevant outputs. The outputs were passed to the functional code at a "task node", as in Denert.

Naturally, the early work in the field did not address the issue of specifying user interfaces that handled direct manipulation specifically. However, Jacob (1986) has shown how the methods that employ state diagrams can accommodate direct manipulation.

Architectures

Following Moran's work, discussed above, much debate has taken place about appropriate architectures for separable user interfaces and a large

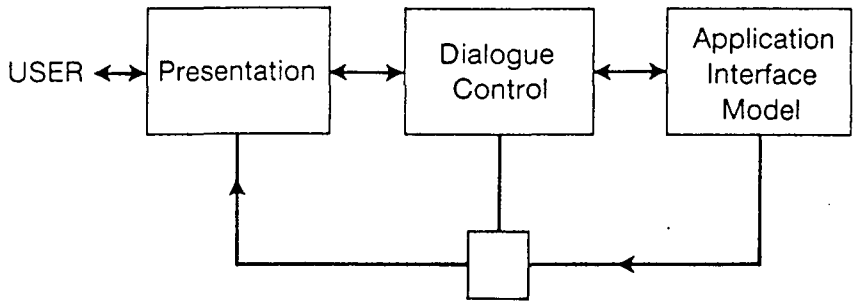


Fig. 1

proportion of the issues of concern have been in the support for separability. In many ways the details of the architecture can be seen as a precise definition of the scope of the user interface.

A very important event in the development of separable user interface architectures was the workshop held in 1983 in Seeheim (Pfaff, 1985). One group in the meeting, consisting of Jan Darksen, Ernest Edmonds, Mark Green, Dan Olsen and Robert Spence, developed an architecture for the user interface (Green, 1985), see Fig. 1.

As Löwgren (1988) points out, however, the first proposal of a user interface software architecture of this form was probably that of Edmonds (1982), see Fig. 2. It built directly upon Moran's concept of layers. In this view the separation between user interface and functional code was precisely at Moran's semantics level where the objects and manipulations that can be seen as atomic, from a user's point of view, reside.

The Seeheim application interface model went further than either Edmonds' or Green's earlier proposals. The idea was that it might contain, for example, information about the effects of operations in order to assist the user interface in the provision of help. Also, in order to deal with larger amounts of data generated by the functional code, such as dynamic images, a direct

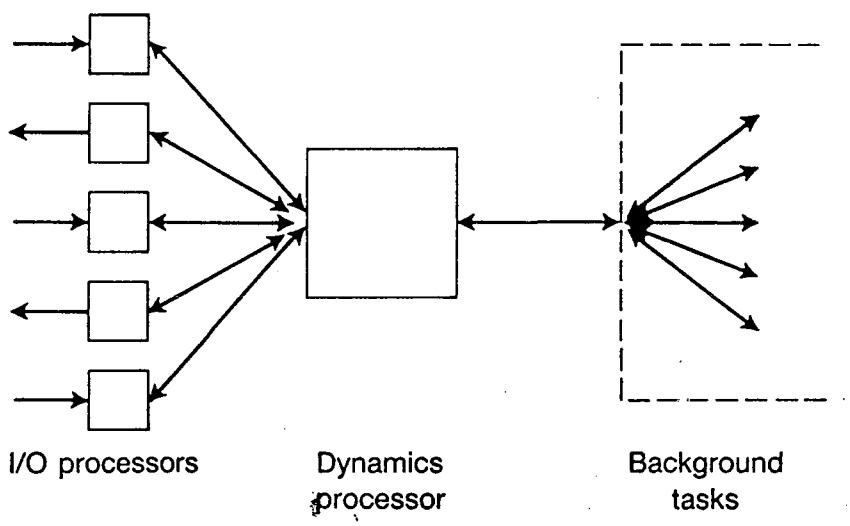


Fig. 2

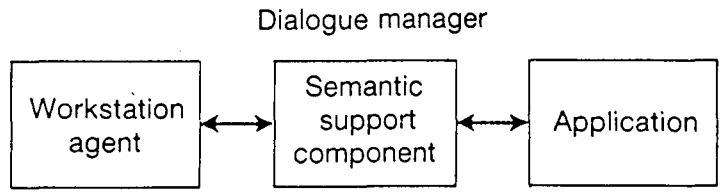


Fig. 3

path to the display was proposed. In this case, the only functions of the user interface were to open such a path and to decide where it should point to on the screen. This architecture has become widely known as *the Seeheim model* and a good deal of the subsequent debate has used it as the base reference. It is, perhaps, not surprising that this model has been so important because, apart from the earlier architecture from Edmonds, position papers were presented at Seeheim by Thomas and Olsen, both of which proposed architectures in the same spirit as what became the Seeheim model.

Many papers have been written concerning modification to this model. For example, Dance et al (1987), see Fig. 3, and Edmonds and McDaid (1990), see Fig. 4. It is noticeable, however, that most are variations on the Seeheim view, rather than radical departures from it. We may assume, therefore, that the climate of opinion remains to favour something of this flavour as an architecture for the separable user interface. In particular, it can be seen that the user interface is seen to include a substantial part of the software of a system.

Practical issues of separability

Certain position papers presented at Seeheim were already addressing practical separability issues. In particular, Olsen (1985) and Sibert et al (1985) consider interdependencies between the layers with some care. One of the working groups, in fact, considered the issue specifically, as reported by

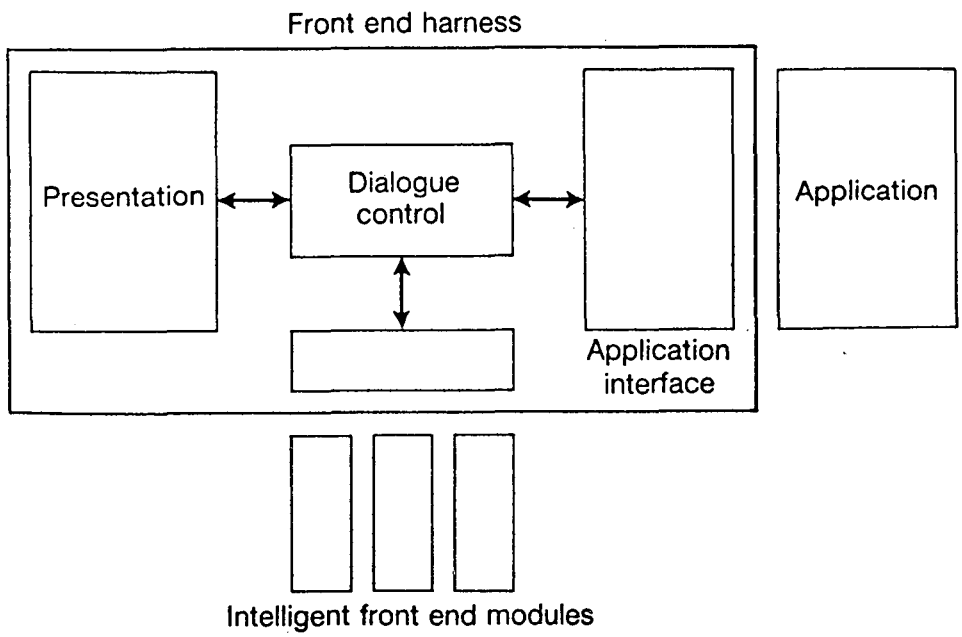


Fig. 4

Enderle (1985). Their basic conclusion was that separability, even for CAD/CAM applications, appeared to be possible given careful design. The group looked particularly at the services provided by the user interface to the functional code and vice versa. This was probably the first attempt to catalogue these difficult concerns (see Table 1).

Szelsky, in his PhD thesis (1988), considered separability in detail. He considered the interface as, perhaps, a slightly shallower entity than others have done, but this enabled him to identify clearly a set of interaction types. This classification is itself a contribution to the definition of the scope of the user interface. Even if one does not wish to limit ones view of the interface in Szelsky's way, it is clear that his work is valuable as a contribution to our understanding of that part of the user interface termed the "presentation layer" at Seeheim.

Alty and McKell (1986) have made a valuable study of the application model in the Seeheim sense. They demonstrate an application model for a command driven system and show how a study of interaction sessions can help to identify the knowledge required in the model. A particular outcome of the empirical study was that they identified a need for a planning component in the application model. Thus, whilst they confirmed the viability of the separable user interface, they demonstrated in practice that the interface can require quite sophisticated components within it.

In another practical case study, Manheimer et al (1989) showed that the separable user interface is a quite realistic concept. It is of interest to note that the mechanisms for coupling user interfaces and functional components vary widely and that, in this case, more than one mechanism was found to be necessary. In applying their LUIS interface system they found the need for both "tightly" and "loosely" coupled relationships. "The tightly coupled routines are linked with the LUIS executable image. A subroutine interface is available for these routines to communicate with the LUIS interface manager, i.e., to send messages to manipulate the ongoing dialogue and to receive messages that inform the application of the state of the dialogue. When a tightly coupled routine is invoked, the user interface is suspended until the application has completed its processing." (Manheimer et al, 1989).

In another practical investigation, Dance et al (1987) encountered the difficulty of rapid semantic feedback in a separable user interface much as discussed at Seeheim (Strubbe, 1985). If one considers a screen displaying three things, a window, an icon for a file and an icon for a disc, then consider the difference between the following two situations. If the user drags the window over the disc icon, the icon is simply obscured. If the user drags the file icon over the disc icon, then the latter is, typically, highlighted. The difference represents more or less instant semantic feedback.

It is sometimes argued that this semantic feedback problem is a case against separability, but in fact that case is far from clear. It is true that it would be

Table 1 Services at the application – UIMS interface

Services of the UIMS for the application	Services of the application for the UIMS
* Give a value matching constraints	* Give a value (e.g., for use in help functions)
* Switch input sources (e.g. between user & journal)	* Give a value to be used instead of user input
* Tell the user a value	* Perform graphics update
* Display some graphics	* Set application state variable to a given value
* Handle an application error	* Grant permission to change application data
* Set ouput context	* Grant permission to change application data
* Set dialogue scope	* Give pick support
	* Give task status
	* (Re-) Set task status to a (previously saved) state
* Undo something	* Undo something

hard to achieve an acceptable performance if tokens had to be passed through the layers, to and from the application, in order to achieve it. However, there is no reason why the presentation layer itself should not contain the appropriate knowledge. After all, the example given above does not imply any application action.

This difficulty may have arisen because many researchers have been influenced by developments in computer graphics where closed packages are often used for implementing interactions. As even the simple example above makes clear, all levels of a separable user interface must be programmable in the sense that the system developer may need to build application specific knowledge into any part of it. This is not an argument against separability, of-course, although it suggests caution about the utility of completely general purpose user interface modules.

The difficulty here is whether the user interface might need too much knowledge (Strubbe, 1985). However, nobody has defined "too much" in this context. Even as early as 1978 Sutton and Sprague found that more than half of the code in interactive business applications was devoted to the user interface. Size is not the problem, although duplicating application code within the interface would be, of course. It would seem that where functional code has to be entered in order to generate the semantic feedback and where that feedback is required very quickly indeed, or where it involves large volumes of data, either the direct path, as proposed at Seeheim, or a closely coupled link, as in Manheimer et al, could be used. However, semantic feedback remains an area for investigation.

Conclusions

Issues of separability still remain to be resolved but there is enough evidence now that the separable user interface can be defined. Perhaps the biggest problem that has stood in the way of successful work in the area is that people have sometimes attempted to build general purpose front ends in the belief that (using the Seeheim terminology) only the application model needs to contain knowledge of application issues. In point of fact, even the presentation layer must often be programmed with specific application dependent knowledge. The separable user interface is not at all ignorant of the functions of the system.

If we return to the definitions of the user interface quoted at the beginning of this paper, it is interesting to notice that the separable user interface can be seen as a concrete elaboration of each of them. Moran (1980) went on to propose that "Any aspect of the system that shows through to the user and enters the user's model is a part of the user interface". The emergence of the separable user interface has occurred partly as an attempt to represent everything "that shows through to the user" in an identified software module. The degree to which this attempt has been successful remains to be fully assessed, but it is quite clear that the study of separable user interfaces is

becoming increasingly important in concentrating our attention on some of the most significant aspects of interactive systems.

Acknowledgement

The preparation of this paper was partly funded by the Alvey programme, grant reference MMI/062, and the ESPRIT2 FOCUS project, reference 2620.

References

ALTY, J.L. & McKELL, P. (1986). Application modelling in a user interface management system. In People & Computers: Designing for Usability. Eds. M.D. Harrison & A.F. Monk. pp. 319–35.

BLACK, J.L. (1977). A general purpose dialogue processor. Proc. Nat. Computer Conf. pp. 397–408.

BOLT, R.A. (1984). The human interface. Lifetime Learning Publications.

DANCE, J.R., GRANOR, T.E., HILL, R.D., HUDSON, S.E., MEADS, J., MYERS, B.A. and SCHULERT, A. (1987). The run-time structure of UIMS-supported applications. Computer Graphics 21, 2, pp. 97–101.

DENERT, E. (1977). Specification and design of dialogue systems with state diagrams. Proc. Int. Computing Symposium, Liège, pp. 417–23.

EDMONDS, E.A. (1974). A process for the development of software for non-technical users as an adaptive system. General Systems, 19, pp. 215–17.

EDMONDS, E.A. & GUEST, S.P. (1977). An interactive tutorial system for teaching programming. IERE Proc. 36 – Computer Systems and Technology, pp. 263–70.

EDMONDS, E.A. (1981). Adaptive man-computer interfaces. In Coombs & Alty (eds.), Computing Skills and the User Interface. Academic Press, pp. 389–426.

EDMONDS, E.A. (1982). The man-computer interface: a note on concepts of design. Int. J. Man-Machine Studies, 16, pp. 231–36.

EDMONDS, E.A. & McDAID, E. (1990). An architecture for knowledge-based front-ends. Knowledge-Based Systems. (To appear).

ENDERLE, G. (1985). Report on the interface of the UIMS to the application. In G.E. Pfaff (ed), User Interface Management Systems. Springer Verlag, pp. 21–9.

FLORENTIN, J.J. (1977). Automatic generation of interactor programs. Proc. Int. Conference on Displays for Man-machine Systems, Lancaster. IEE pub. no. 150, pp. 126–9.

GREEN, M. (1981). A methodology for the specification of graphical user interface. Computer Graphics, 15, 3, pp. 99–109.

GREEN, M. (1985). Report on dialogue specification tools. In G.E. Pfaff (Ed), User Interface Management Systems, Spring Verlag, pp. 9–20.

HEKMANPOUR, S. & INCE, D.C. (1987). Evolutionary prototyping and the human-computer interface. In: H.J. Bullinger & B. Shackel, eds. Proc. of INTERACT 87. Amsterdam: North Holland; IFIP, pp. 479–484.

JACOB, R.J.K. (1986). A specification language for direct manipulation user interfaces. ACM Transaction on Graphics. 5, 4, pp. 283–317.

LAFUENTE, J.M. & GRIES, D. (1978). Language facilities for programming user-computer dialogues. IBM J Res. Develop. 22, 2, pp. 145–58.

LÖWGREN, J. (1988). History, state and future of user interface management systems. SIGCHI Bulletin, 20, 1, pp. 32–44.

MAHER, P.K.C. & BELL, H.V. (1977). The man machine interface – a new approach. Proc. Displays for Man-Machine Systems. IEE Publication 150, pp. 122–5.

MANHEIMER, J.M., BURNETT, R.C. & WALLERS, J.A. (1989). A case study of user interface management system development and application. Proc. CHI'89, pp. 127–32.

MORAN, T. (1980). A framework for studying human-computer interaction. In Guedj, R.A., Ten Hagen, P.J.W., Hopgood, F.R.A., Tucker, H.A. & Duce, D.A., Eds., Methodology of Interaction, Amsterdam: North-Holland, pp. 293–302.

MORAN, T. (1981). The command language grammar, a representation for the user interface of interactive computer systems. Int. J. Man-Machine Studies, 15, pp. 3–50.
NEWMAN, W. (1968a). A graphical technique for numerical input. Computer Journal, 11, pp. 63–4.
NEWMAN, W. (1968b). A system for interactive graphical programming. Proc. of Spring Joint Computer Conference, pp. 47–54.
NEWMAN, I.A. (1978). Personalised user interfaces to computer systems. Proc. of the European Computing Congress, London, pp. 473–86.
OLSEN, D.R. (1985). Presentational, syntactic and semantic components of interactive dialogue specifications. In G.E. Pfaff (Ed.), User Interface Management Systems. Springer Verlag, pp. 125–33.
PARNAS, D.L. (1969). On the use of transition diagrams in the design of a computer interface for an interactive computer system. Proc. Nat. ACM Conference, pp. 379–85.
PFAFF, G.E. (Ed.) (1985). User interface management systems. Proc. of the Workshop on User Interface Management Systems, Seeheim, 1983. Springer Verlag.
SHNEIDERMAN, B. (1987). Designing the user interface. Addison-Wesley.
SIBERT, J., BELLIARDI, R. and KAMRAN, A. (1985). Some thoughts on the Interface Between User Interface Management Systems and Application Software. In. G.E. Pfaff (Ed.), User Interface Management Systems. Springer Verlag, pp. 183–92.
STRUBBE, H.J. (1985). Report on role, model, structure and construction of a UIMS. In G.E. Pfaff (Ed.), User Interface Management Systems. Springer Verlag, pp. 3–8.
SUTHERLAND, I.E. (1965). SKETCHPAD: A man-machine graphical communication system. MIT Lincoln Lab, Lexington, Mass. Tech. Report TR-296.
STALLING, W. (1974). Some aspects of an interactive dialogue-generation facility. Proc. Intl. Conf. on Systems, Man and Cybernetics, pp. 495–498.
SUTTON, J. and SPRAGUE, R. (1978). A study of display generation and management in interactive business applications. Tech. Report RJ2392 (31804). IBM San José Research Laboratory.
SZEKELY, P. (1988). Separating the user interface from the functionality of application programs. CMU-CS-88-101. Carnegie-Mellon University.

PROTEUS: Collecting data about the experience of using an interface

Jonathan Mark Crellin, School of Computing and Information Technology, University of Wolverhampton.

Abstract: This paper describes current problems in interface evaluation, and proposes a solution to those problems in PROTEUS, an interface evaluation technique. Problems stem from the differences in perception of an interface design between designers of systems and users of systems. An expert novice perspective can be used to consider this problem. Areas that can provide possible solutions to the problem are knowledge elicitation, personal construct psychology, and prototyping. These approaches have been combined in PROTEUS. The paper then describes empirical work on PROTEUS and continues to consider the future directions that the development of PROTEUS may take.

The problem...

There are many different ways to characterise interface issues. One useful approach has been taken by Marcus. Marcus suggests that communication between people and computers takes place in at least three phases that represent three different "faces" of computers. These faces consist of different conceptual frames for communication. **Outerfaces** are those parts of the communication that are the final products of the computation. Typically they consist of the text, diagrams, graphics, that may be displayed on the computer screen, or printed on paper. The people who use these products of the computer do not need to have very much knowledge of computers or the means of display to understand the communication embodied by these products. The outerface of the HCI is independent of the media. It is these products that are the goal of the user's task. The **interface** is the aspect of the human computer communication that is concerned with command, control and documentation. This face provides the handle for the computer as an artefact. Without these controls the computer would be useless. The sorts of people who encounter computer interfaces can vary tremendously in their prior experience of computer systems. The final face of the computer is the **innerface**. The innerface consists of the parts of the software that only the system designer confronts. These parts control how the system functions, that only the people who build or maintain the system have to interact with. They consist of programming languages, software tools, and operating systems used to develop the system. They are in effect the 'cogs and gears' of the computer system Marcus, 1983.

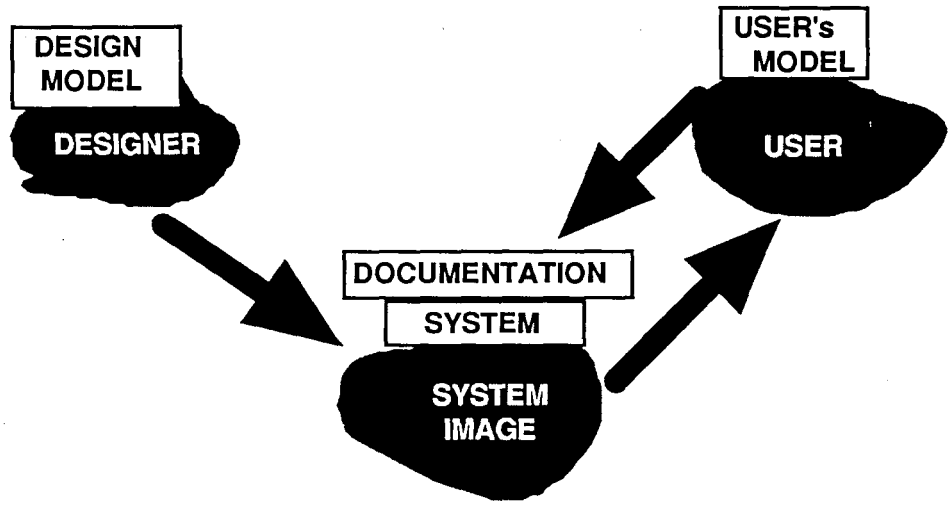
There is a major difference between users of computer systems and designers of computer systems in the way that they relate to these different faces. Designers consider first the innerface, then the interface and outerface. They initially make decisions about what functions are feasible (innerface), and then how these are to be controlled by the user (interface) and how they will be expressed as output (outerface). Users of computer systems are primarily concerned with the products of the interaction (outerface), that are also the goals of the task the computer system supports. The controls for the system (interface) are unavoidable for users. Most users hope the interface is invisible, with the *task* structure forming the perceived structure of the interface, rather than the structure of the software. Users wish to have nothing to do with the innerface of the system. If forced to interact with the innerface (for example if the system crashes) most users will give up the task, or seek help from elsewhere.

This difference between designers and users goes some way to illuminating a central problem of design in HCI. Designers focus on the computer system. User focus on the task domain the system supports. Writers use word processors to produce text, graphic designers use desk top publishing packages to produce page layouts. It is easy for the computer system designer to support those aspects of the task that are easily simplified by a computer

system, at the expense of those aspects of the task that do not easily map to a computer based system.

Designers and users have different knowledge about computer systems, and about the task domain. They also have different backgrounds that leads to different ideas about how problems should be solved. This different culture also leads designers and users to have a different language about HCIs. Users' ideas are derived from their experience as users of different systems, and knowledge about the application area. Designers' ideas are derived from their background in software engineering, mediated by training in human factors and the design of HCIs, and in task analysis. Many problems of computer systems can be traced to an inadequate understanding by computer designers of how users understand their tasks and how these tasks are represented by a computer system.

When a user wants to do something with a computer system he will need to plan the steps necessary to perform a particular action. This planning is based on the users' model of the system. The more effective a user's understanding of the physical system the more likely this plan is to work. Norman points out that there are at least three different components that make up the designer/user model system. The first of these is the **design model**, that precedes the actual system, and represents the actual system that is constructed. The system when built projects an image about itself. This image is derived from both the user interface to the system, and the system documentation. From this **system image** is derived the **users model**. Users' develop their model based on their experience of the system. The user model must accurately reflect the design model of the system if the user is to plan actions on the system effectively, and this can only happen through the mediation of the system image. In the design of systems the designer will also have a model about a typical user, that forms a part of the design model, Norman, 1986.



Mental models and components of the Computer/Designer/User System (adapted from Norman, 1986).

When any of these models or images do not match with each other, or with the system, then usability problems are likely to occur. Therefore elicitation of users' knowledge of systems can be appropriate. This information can be compared to both the physical characteristics of the system, and to the design model.

Various techniques have been developed to address this problem. The following section looks at approaches that have been taken to the design of human computer interfaces, focusing on how differences in designers' and users' perspectives are dealt with.

Approaches to the Design of Human Computer Interfaces

Interface design involves co-ordinating knowledge from a number of different domains. Software engineering knowledge defines what is possible within the technology. Cognitive psychology provides information about how people process information, and the capacity of

various cognitive stores and registers. Aesthetics and design provide information about what people enjoy, and find pleasing. Task descriptions involve understanding the actual activities that the system is designed to support. Contextual and social factors involve understanding how different tasks are likely to be performed, when users are doing real work on the system, in a real workplace.

These different forms of knowledge are used at different points in the design of a system. Several design and evaluation methods exploit these different forms of knowledge. In early computer systems the focus was driven by what was possible within the technology. Most users of systems had a computing background, and shared a considerable amount knowledge with the system designers. The development of single chip computers has considerably reduced the price of computers, and they are now used to support many more tasks, by a larger, but less computer literate population. Commercial pressures alone have driven software houses to be much more concerned to design systems that meet human cognitive and affectational requirements, and that adequately support tasks. Failures in the design of interfaces has lead to more evaluation of contextual and social aspects of computer use.

Design is a process that involves creation of new structures, within the context of pre-existing ones. The final design is rarely conceived in any complete form at the beginning of the design process, if such were to occur it would probably indicate premature focus on a solution. The process involves initial definition of the problem, concept invention, decision-making when two alternatives present themselves, and evaluation of the design as it emerges (formative evaluation). This evaluation stage is useful in deciding which alternative design to develop. In HCI design the importance of understanding how users will react to a proposed design requires that some form of user representation is involved in the design process. That user representation can be in the form of a conceptual model, or real users performing tasks on prototypes. Guidelines contain implicit conceptual models of human performance. Analytic methods use similar (but often rather one dimensional) models of human performance. The huge variety of tasks that computer systems are now used to support results in unanticipated variations in performance. Empirical methods particularly informal empirical techniques, are more likely to detect unanticipated issues. Experimental, survey and contextual or observational methods all employ real users, although the immediacy of the data collected, and the ecological validity of the environment in that the data is collected varies between these three methods. Design methods are currently being developed to address some of these issues (for example the PICTIVE method, Muller, 1991).

Characteristics of an improved method of evaluation

The problem identified is that the differences between designers' and users' knowledge degrades communication between them, and this results in software designs that users find hard to understand and use. Therefore important characteristics for design/evaluation methods ought to include the following

- Employs real users rather than representations of users
- Can be employed in realistic contexts
- Does not focus on designers' issues, but on users' issues
- Promotes communication between users and designers
- Discourages early focus on a solution
- Encourages designers to view software from a users' point of view

This paper describes a dynamic conflict. Coping with individuals' needs requires a method of collecting the uniquely individual subjective realities that form the basis of each person's understanding of the world. In contrast design involves construction of a single solution to a

problem, and this implies an underlying shared objective reality that the design taps into. The meeting point of these two perspectives is the shared reality of the designer and user. This shared reality is ultimately the 'objective' reality that the design is built to meet. The tools described in this paper help make explicit the features of that shared reality, both to designers and to users, and help the development of systems that reflect user needs.

Prototyping and sketching

Prototyping is a term that has been employed in a variety of different ways in different manufacturing contexts. In its original context of engineered artefact manufacturing, a prototype is a hand-built version of the artefact. Such an artefact may be anything from a fighter aircraft, to a ball point pen. Craft techniques are used to manufacture the artefact, and possibly different materials are used for certain components (for example turned wood instead of moulded plastic). The prototype is fully functional, and made to the same tolerances as those intended for the final product. The prototype may be evaluated on a number of criteria. Its performance is measured, its usability may be evaluated, and the cost of manufacturing components estimated. From the prototype jigs, moulds and cutting tools can be prepared, and the mass-production run of the final form can then take place. This form of prototype is concerned with establishing that it is possible to implement the design, that it will perform as expected, and that conventional manufacturing processes can be used for its construction within allowable costs. Such prototypes are sometimes referred to as pre-production prototypes.

In software engineering prototyping usually has a much less precise meaning. There are at least two prevailing ideas about what a prototype should be. An engineering perspective suggests that a prototype should be a near complete system, including much of the final systems functionality and user interface. A more craft-like perspective suggests that prototypes can be rough preparatory representations of parts of the final system, and that often multiple prototypes (which enable comparisons to be made) are more useful than a single definitive prototype. A software system can be divided into the functional code, which performs the functions of the system, and the human computer interface, with which the user interacts. During the development of a system it is useful to evaluate different parts of the system in isolation. Therefore the term 'prototype' has often been used to describe non-functional representations of systems. There is argument about the applicability of the term 'prototype' to these non-functional, preparatory representations of the system. Gregory suggests that in this context the term mockup may lead to less misunderstanding. A mockup is a non-functional representation of a system, or part of a system. Gregory argues that the term prototype is most properly used to describe a functional (although possibly imperfect) version of the final system (Gregory, 1984).

Hekmatpour and Ince suggest that a prototype will have the following characteristics (Hekmatpour and Ince, 1988):-

- It is a system that actually works.
- It will not have a generalised lifetime (it could be thrown away, or form the basis for final product).
- It may serve many purposes.
- It can be quickly and cheaply built.
- It forms an integral part of iterative process also including modification and evaluation.

They point out that in software engineering there is no mass-production phase in the development process. The pre-production prototype is also the final product. Once the coding has been completed the software can be duplicated in any required numbers. In this sense software development is more closely allied to craft production than manufacturing. In craft production a single object is made, in manufacturing large numbers of objects of a single design are made. Prototypes are used in the development and evaluation of the design of the system. Hekmatpour and Ince also refer to three forms that prototypes can take. The first is an imperfect but fully functional version of the system. The second is a 'bread-board'

system that is fully functional, but lacks a user interface, and the third is a full user interface which lacks most of the final systems functionality. Another way of classifying prototypes is into three types that highlight the different development roles of prototypes:-

- Throw-it-away prototypes: This type of prototype is used for requirements capture and specification. It needs rapid development, and may be implemented using other hardware environments to that used for the final product. Non of the code in the prototype is used in the final product. Such an approach is also sometimes referred to as rapid prototyping.
- Evolutionary prototypes: The purpose of an evolutionary prototype is to introduce a system into an organisation gradually and flexibly to allow changes as problems occur, there is never a 'final' product. Different parts of the system may be implemented at different times.
- Incremental Prototyping: The system is built incrementally, one section at a time, however focused on one overall final design (hence different from the above).

This paper will focus on the 'throw-away' prototype, as this best embodies the exploratory prototyping encouraged by PROTEUS. A throw-away prototype is a system implemented using an alternative development system to that used for the final version of the system. Such alternative development systems usually allow development in a very short time, but at the cost of limitations in the prototype. For example, neither the size of the code, nor the speed of program execution, nor the hardware on which the prototype has to operate, may be optimal. The 'throw-away' of the title is because little or non of the code that implements the prototype is used in the final system. Only the design is retained.

Sketching

Prototyping can be employed during the design process in two different ways. First to communicate ideas within the design team, where evaluation will be mostly of the heuristic variety (Nielsen and Molich, 1990); and second for user evaluation, where prototypes can be used by sample user populations, and evaluations collected. In this case the empirical evaluation could be either 'observational', 'experimental' or 'survey' (Long and Whitefield, 1986).

Prototyping can be used in system design in a similar way to the use of sketching in graphic design. Alternative solutions to a particular problem can be implemented, as a concrete focus for decision making during the design process. Within the PROTEUS methodology prototypes are used to communicate what a designer considers to be the range of alternatives that may be possible. The design 'solution space' is described by such prototypes. Obviously such an approach is partially derived from the architect's concept of preparatory sketches Graves, 1976, that indicate different possibilities for the final design.

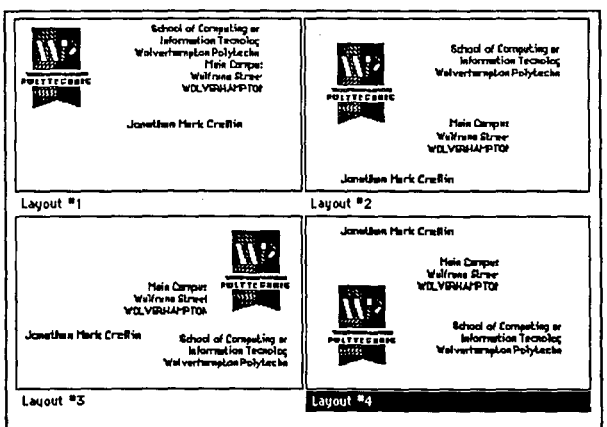
Prototyping and Design

A simple approach to software design might be that as users' know their requirements best, it is they who should design the software tools that they require. However simply making easy to use tools (for example HyperCard) available to novice designers, (experts in the target domain, but not expert in system design), is not a solution to system design. (Even HyperCard takes two days to learn to use, Thovtrup and Nielsen, 1991). A system designer also needs to understand what is possible and what is not possible within a human computer interface, and to have knowledge about how people interact with systems. The problems of making design development tools available to people who are inexperienced in design is highlighted by problems experienced in different areas of design. Anecdotal problems experienced by amateur graphic designers, using desk-top publishing packages to produce documents have been well reported. Graphic design guidelines (such as using a single type face (in Windowsese a font) for body text for example) are often ignored, and familiarity with the document, and the finished appearance of the laser printed page conceals the illegibility of the design from the amateur designer. What is needed is an awareness of graphic design guidelines, the freedom to consider design alternatives, as well as a tool that enables a novice to produce a finished artefact. Early (and inappropriate) focus on a solution is a particular problem experienced by novices.

Computer systems as design tools frequently fail to provide the richness necessary for exploratory representations of a design solution. Hewson has pointed to the way current computerised graphic design packages fail to provide provisional representations of solutions (Hewson, 1990). Much of the early work of a graphic designer is concerned with exploring ideas, and quickly representing these (thumbnail sketches) using a variety of graphic media (Martin et al refer to similar observations concerning VLSI designers using CAD systems Martin, et al., 1990). The range of the graphic media (different types of pencil, pens, and paper) allow an almost infinitely large variety of line and tonal quality to be created quickly, which can in turn be used to represent abstractions of the final design. Pencil lines of varying width and weight can be used to represent different type faces. Sketching allows these provisional representations are explored during the design process.

Graves refers to three forms of sketching: referential; preparatory; and definitive, which have different functions in the practice of architectural design. Referential sketching involves collecting visual experiences, it is easier to recall things once they have been cognitively processed during the process of sketching. Preparatory sketching involves trying out a variety of different ideas, quickly and informally. Definitive sketching is the process of representing in detail what the final design solution should be (Graves, 1976). The last form of sketching is well supported during the interface design process. A highly finished representation of a solution can be generated very quickly using a computer system. It is the first two forms of sketching that are poorly supported in computer aided design of most types. Novices can produce very finished results very quickly using computer packages. Unfortunately the very finished quality of computer products frequently gives them a spurious authority, and stops further exploration and improvement. In computer aided design the ease of definitive sketching suppresses preparatory sketching. Similar problems are experienced in interface design. A Visual Basic interface, constructed of familiar and professional looking Windows features (buttons, icons and graphics), frequently belies the underlying difficulties of the overall design. When it is easy to construct the final form for a system, it is very easy to rush straight to the first solution that is thought of.

Recent computerised design tools attempt to resolve some of these problems by creating a variety of solutions to a particular job, using the elements that a user has specified. Multi-ad Creator, in the domain of graphic design, generates alternative layouts for a single page document, using the objects (blocks of text, headings or graphics) that have been created by the user. The program responds to general design guidelines, and can be adjusted to emphasise text or graphic objects (StudioBoxLtd., 1990).



Screen shot from multi-ad creator, showing automatically generated alternative layouts for a piece of graphic design.

In this way the user is forced into considering alternative solutions for a given specification. There is a difficulty in finding the appropriate 'sketching' media for developing ideas about interfaces. Conventional sketching is suitable for graphic or architectural design problems, as both these are essentially visual. (Buildings are primarily perceived through the sense of vision.) Traditional sketching is less useful for interface design which involves the process of perceiving two dimensional images which change over time. Another possibility for representing interfaces is story boarding. Story boards are used in film making, and expresses the linkages between different screen images. However both these possibilities do not adequately express the highly interactive, branching qualities of human computer interfaces.

PROTEUS and Prototyping

In PROTEUS it is possible to evaluate both fully functional prototypes, and mockups, or partial systems. Prototyping is involved in PROTEUS in several ways. First, prototypes are used for of communication of a designer's ideas about possible individual designs. Second, multiple prototypes define the design space, showing the range of alternatives that the designer thinks may be appropriate. The prototypes are never judged in isolation, but as comparisons. Users are likely to lack a framework in which to evaluate single systems. Their experience of alternative systems is likely to be limited. It is only by presented more than one system and forcing comparison that realistic and reliable evaluations by users can take place. The use of multiple prototypes increases the workload on designers, but this is obviated to some extent by the increased availability of tools for building prototypes quickly.

Finally the range of prototypes that can be used is from simple, non functional representations of the system, to complex fully functional prototypes. The experimental work described below has involved using both a series of fully functional prototypes, (in the interface evaluation study), and a series of partial systems (in the RAS help system study). The methodology supports evaluation of components of an interface design (for example alternative screen designs), or fully functional prototypes. A minimal number of prototypes should be presented to users in order to support the idea that the prototypes represent the design space.

Personal Construct Psychology

Personal Construct Psychology (PCP) is an explanation of how people think, that uses the metaphor of the 'Personal Scientist'. It suggests that people engage in a process of understanding the world by forming theories about how it works. These 'theories' are retained as long as they work, and provide useful predictions about the world. Eventually these theories are replaced when the number of contradictions between the theory and the experience of the world becomes unbearable, and alternative an theory can be found that performs more effectively. The theory behind Personal Construct Psychology was developed by George Kelly in 1955 (Kelly, 1955), and was originally applied in a teaching and then a therapeutic context. It has been applied to a number of domains since 1955, including marketing and business applications, knowledge elicitation, and now interface design (Frost and Braine, 1967, Stewart and Stewart, 1981, Boose, 1985, Crellin, 1988, Crellin, 1989). One of the spin-offs from Personal Construct Psychology is a tool for exploring 'cognitive space' called the Repertory Grid (sometimes called the Rep Grid or the Kelly Grid). The repertory grid collects the ways people distinguish between objects in their world, the theories they have about what is important about the objects they deal with. These 'objects' could be anything, in a therapeutic context they might be people, or roles of people in a person's family. In an advertisng context they might be different brands of a particular type of product. In PROTEUS the objects are different designs for an interface.

One of the advantages of the repertory grid is that it is an easy way of collecting relatively rich, open-ended qualitative data, that is still fairly structured, so that some comparison can be made between different individual's ways of understanding their environment. It is also a tool that is fairly easy to automate, enabling the collection of qualitative data without the usual overhead of time associated with most qualitative data collection techniques (Shaw, 1980).

Initial Development of the Construct Elicitation System

The idea of using the repertory grid for collecting data about user interface designs was initially tried with a combination of manual repertory grid elicitation, and available repertory grid elicitation and analysis software. In particular an early study used the Construct and Grid Elicitation system (CAGE) that was used on Open University psychology courses. This involved collecting repertory grid data about a range of word processing systems that had been used at the Open University during the previous few years. The study was reasonably successful, in that it demonstrated that repertory grid data could be elicited about such a set of elements, and also that the information collected did give some insight into the ways user of the systems thought about the pertinent characteristics of the systems. The study also highlighted the need for a customised software tool to support this technique.

As a result, the initial specifications for the Construct Elicitation System (CES) component of PROTEUS were that it should elicit constructs from users interactively, and store these for later analysis. The elicitation should be user event driven rather than using a system directed dialogue (experience with CAGE suggested this). Modality should be avoided where possible.

First Prototype for Construct Elicitation System

The failure of existing repertory grid based software to satisfy requirements for a usability evaluation system lead to the development of an interactive elicitation tool. The first prototype concentrated on grid elicitation, with analysis of the resulting grid being performed by a version of Mildred Shaw's FOCUS program (Shaw, 1980, Shaw, 1984), (essentially a set of sub-routines from within Shaw's PEGASUS program). The characteristics of the system were that it was a menu driven interface, that loaded a data file containing element names, then elicited constructs interactively using triadic elicitation. Particular care was taken in the wording of the prompts relating to the personal construct psychology (PCP) task, as experience with the CAGE system showed these were the most likely to be misinterpreted.

Verbal Protocol study

This test bed system was used for early usability evaluation with two subjects employing a verbal protocol methodology, and a Moves Goals and Sub-goals type of evaluation, following (Ericsson and Simon, 1984, Robson and Crellin, 1989). A 'verbal protocol' version of the programme was devised to encourage a running dialogue by the user. This included an additional prompt, asking for users to say aloud why they chose a particular option. (Koubeck and his colleagues describe a similar method of obtaining a computer aided verbal protocol Koubeck, et al., 1987.) This prompt was placed in the sub-routine that checked user input for the menu command, so was not displayed after every user input. (At certain places in the program the user was already committed to a course of action so was not allowed to abort a particular action by returning to the menu.) On these occasions the verbal protocol prompt was not displayed.

Conclusions

The highest level goal in using PROTEUS is to generate a representation of the way in which the user of the system discriminates between elements. This was not always clear to users of the prototype system. The task was fairly interesting to complete, but lacked a focus because subjects could not see what the final product was, or appreciate the effect of each elicited construct on that product. This certainly contributed to a limitation in the number of constructs elicited. The feedback system (PEGASUS) developed by Shaw, 1978, makes users aware of close matches between elements and constructs. However experience as a user of this system suggests that this is not easy to integrate into a whole picture of the emerging representation of the different elements. The method employed by Boose, 1985 in ETS of displaying the resulting relationship between elements (used in the context of extracting expert knowledge) seems more appropriate to the task of producing a representation.

The way constructs were elicited remained a crucial bottleneck in the prototype. It was the most difficult feature for users to conceptualise correctly. This evaluation established several important issues for future developments of the Construct Elicitation System. The nature of the task was not very clear to users, the verbal-protocol task was confused with the personal construct task on several occasions. Navigation through the system proved difficult. These issues lead to the porting of the program to a Graphic User Interface (GUI) using an Apple Macintosh. This considerably eased some of the problems of using the menu driven interface in the prototype. In particular the system assumed a far more event driven character than in the earlier environment.

Further development of the Construct Elicitation System

The system (as implemented on the Macintosh) had the following characteristics. The program still cycled through triadic elicitation, but system operations were made through

Macintosh pull-down menus. Feedback was made available as a similarities matrix, providing similarities between elements, and later by using binary trees to represent the FOCUS analysis of repertory grid data. Users were then asked if they wished to discriminate between closely matched elements, which they could do by adding discriminating constructs. If the system was to be used to discriminate between different interfaces then the system should run on the same machine as the 'target' systems, and needs to be small enough to fit on any machine likely to be used, so that the CES task could be performed in a user's real working environment.

Report Writer and the Analysis of Repertory Grid Data

The FOCUS algorithm is the most commonly used method for analysing repertory grid. The FOCUS algorithm can be displayed as a binary tree. This is as intuitive a way as any of communicating the important features of a cluster analysis. The existing software to perform this type of analysis was not well suited to the needs of this project. A new package (called Report Writer) was developed, initially as a separate module from the Construct Elicitation System, and subsequently incorporated into the CES. Report Writer can produce tree diagrams on screen, or on any available printer. Correlation tables (which are required to read the binary trees accurately) can also be printed to screen or hard copy. The data can then be analysed by an experienced researcher using a process of progressive abstraction. Ethnographic researchers are generally not concerned with quantification or statistics, but rather with understanding events as they happen Davies, 1990. This usually involves grappling with huge amounts of data, then abstracting down to fewer and fewer abstractions that give a meaningful perspective on what is going on. A sequence of abstraction given by one researcher quoted by Davies is as follows:-

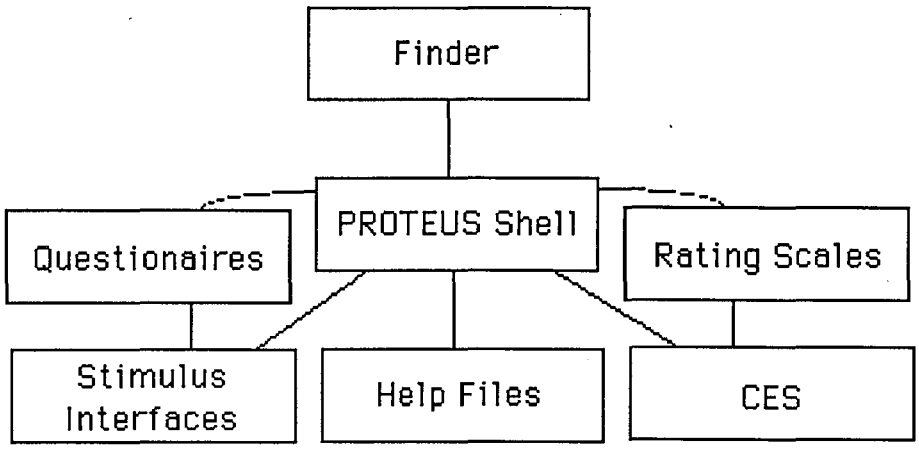
300 Abstractions --> 30 --> 10 Main Themes --> 6 (for write up).

Analysis of the Construct Elicitation System follows a similar model. Initially each individual's constructs can be regarded as abstractions, representations of the ways that individual uses to understand what is going on. Cluster analysis provides a way of looking at the differences between the different constructs, and a first level of abstraction. It can become clear from the cluster analysis how many underlying abstractions there are, which reflect the construct data. This allows the analyst to develop an overview of the important characteristics of the interface for a particular individual. Subsequently different individuals can be compared, and a general hypothesis for perception of the different interfaces developed. As a general rule most people have two or three ways of looking at the different interfaces, although in some cases there is only one predominant way of looking at the interfaces, and in some cases there are many dissociated ways of describing the interfaces.

The Report Writer provides a useful first step in the analysis of data, but the successful analysis of repertory grid data requires that information from a variety of sources be brought together. Correlation tables need to be combined with tree diagrams, and the textual content of constructs, to form appropriate abstractions from the data easily. At the moment these different forms of data are available separately, and therefore analysis involves the shuffling of a series of printouts. Hypertext methods of presenting data may be particularly appropriate.

Integration of the Construct Elicitation System into a stable environment

A shell system (PROTEUS) was constructed to control and monitor user behaviour. This allows users to move between different parts of the evaluation task easily. The use of a shell facilitates monitoring of user behaviour, allows easy access to help screens, and can control the delivery of questionnaire and rating scales to the users. The different parts of the task can be given separate identities, helping navigation around the relatively complex evaluation environment. Finally the shell increases the robustness of the evaluation system, to the extent that it can be simply given to users, for use on their own machines, in ecologically valid settings.

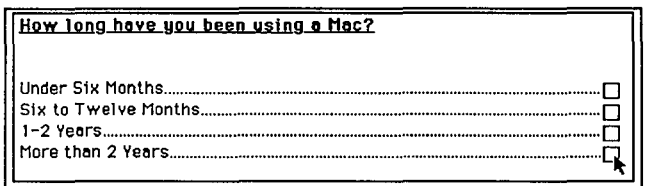


Structure of PROTEUS

The system was tested using several stimulus interfaces for the Interface Evaluation Study (IES). A small trial study preceded the Interface Evaluation Study. Data from the trial lead to a few minor changes in procedure of the IES, and to modification and extension of the Instructions to Subjects. Although subjects can leave CES and re-examine the stimulus interfaces (via the PROTEUS shell) this occasionally proved too time consuming and disruptive, especially during the difficult construct elicitation task.

Current State of Development of CES and PROTEUS

The following section describes the current state of development of the PROTUS software tools. PROTEUS consists of several components that help in the evaluation of software prototypes by users, and support understanding of the data generated from users, by designers. The PROTEUS shell program governs the use of prototypes, questionnaire and rating programs, the Construct Elicitation System, and of help files. It also provides data on system usage. It is set up by a configuration file that contains the names of all the files to be used by PROTEUS.

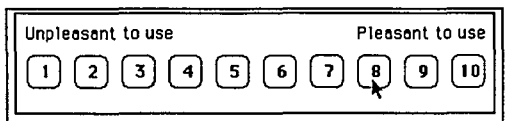


Screen shot of questionnaire screen

PROTEUS provides a screen of information to users on their usage of the different prototypes, by filling a pie chart (stimulus interface log). This allows users to distribute their time fairly between the different prototypes. The pull down menus available in this screen allow access to the help files (which are simple text files), and to quit to finder. Initially a user can only select prototypes, but after all prototypes have been used once it becomes possible to select CES.

The questionnaire program is called by PROTEUS, and is used to collect demographic data and information about prior experience from the user. The program supports branched multiple choice questions, each with a single response. Questions are presented on the computer screen, and the response is selected by the user using mouse and radio-buttons.

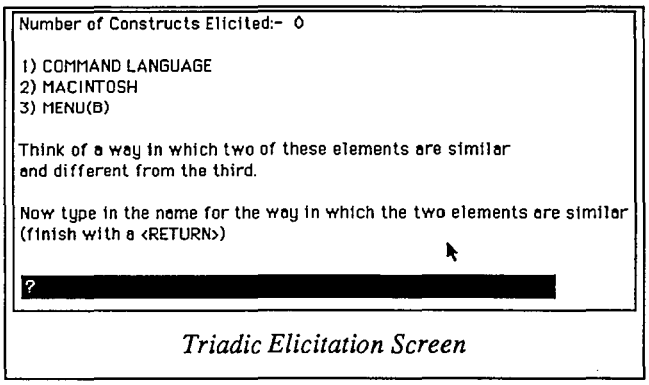
Ratings data is collected from users by a program before first use of CES, and after last use. The program asks for ratings on a ten point scale, for each of the prototypes. This gives an indication of changes in perception (if any) that result from the repertory grid task. PCP theory



Screen shot of rating screen

suggests that constructs are very dynamic, and are likely to change with an individuals experience. Fortunately the early studies show that in the interface design context ratings of the stimulus interfaces rarely changes.

CES is a complex program, with many different screens. Construct elicitation (of triadic or implication type) is presented on start up of the application. Responses are collected in a single line edit field. Ratings for constructs are collected by mouse selection from a seven point scale.

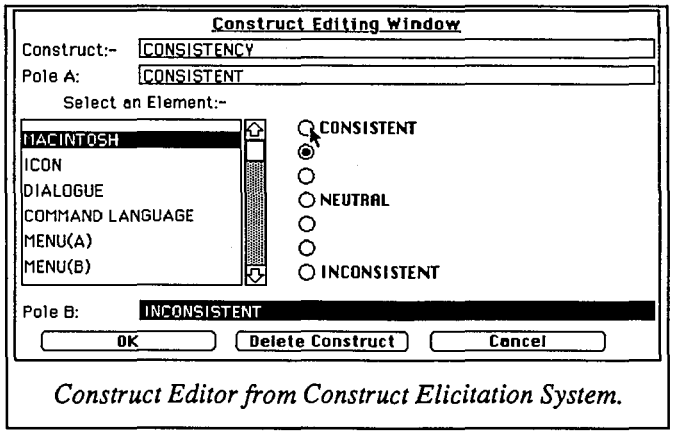


Triadic Elicitation Screen

The construct editor allows all construct attributes to be modified. This includes the main construct name, pole names and ratings for any or all elements. Additionally the construct can be deleted. Before users can leave the system they are asked to provide extended descriptions of constructs. These are collected in a text edit screen, for each construct. Users have the option to go forwards or backwards between the constructs, and also to undo any changes they have made to the text descriptions.

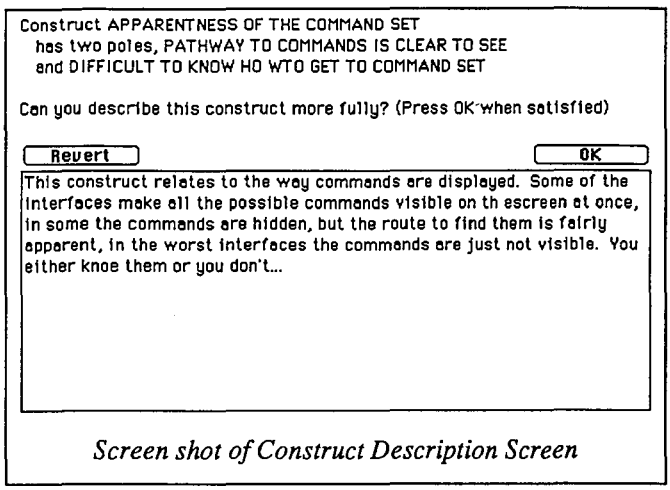
Finally it is possible for users to see the similarity ratings between elements. This is called up from the elements menu and provides the similarity ratings in tabular form, and a binary tree representation of the FOCUS analysis. Closely matched elements are highlighted to the user, and a course of action to increase differentiation suggested as a *possible* next step. Users may feel that the similarity between two elements reflects what they feel about the element set, in which case they do not need to add in extra discriminating constructs.

CES can be used as an independent application (as it was in the NPL study, Crellin, 1988), or it can be used as a component of the PROTEUS shell.



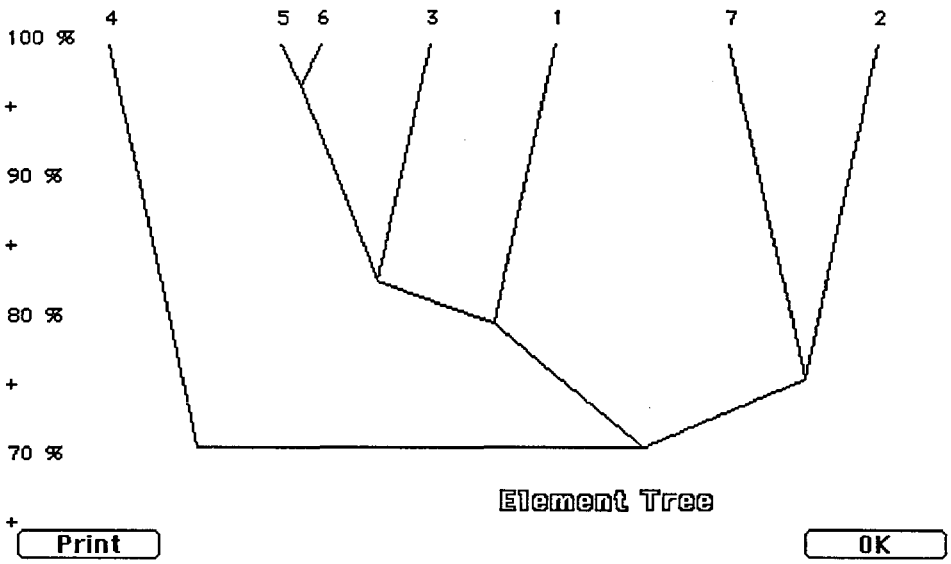
Construct Editor from Construct Elicitation System.

Report Writer is an application for the FOCUS analysis of repertory grids. It can perform analysis of grid of up to thirty elements and constructs. Output from the program is as similarity tables for elements or constructs, and as binary tree diagrams showing the clustering of the elements or constructs. Output from the program is either to screen or printer.



Screen shot of Construct Description Screen

Report Writer will load information in any of the formats used during the development of CES. It can also be used to concatenate constructs from several peoples constructs (where the same set of elements have been used) to evaluate the relationship between those constructs.



Binary Tree Representation of Element Clustering from Report Writer

Empirical validation

A number of evaluation studies took place during the development of PROTEUS. These studies covered a range of different issues within the prototyping of design solutions within the interface design space. Initial prototypes used were full prototypes of a limited system (the Interface Evaluation Study used seven full and very differentiated prototypes to a simple system, described in Crellin, 1989 and Crellin, 1992. Later studies looked at prototypes which were vertical prototypes, testing a sub-system of a final system, (Crellin, 1990). Different ways of relating the prototypes to PROTEUS were also examined. These ranged from the optimal setup, where prototypes were accessed through the PROTEUS shell, so that other information about usage of the system can be recorded, as a sub-system, implemented on a different hardware base, that simulated the interaction of the target system (a Macintosh simulation of a command line interface) (Crellin, et al., 1990, Crellin, 1992).

Comparison with other evaluation methods

A comparison of PROTEUS and other evaluation techniques was made, using the original seven prototype interfaces (Crellin, et al., 1990, Crellin and Preece, 1991). This study took place in a small commercial company. This comparison looked at formal analytical techniques (keystroke-level analysis, Card, et al., 1980, and Reisner's BNF, Reisner, 1981) and verbal, video and software log protocols to collect informal usability information, in an

environment close to users' normal working environment, following Whiteside's description of contextual evaluation (Whiteside, et al., 1988). The results of this study showed the importance of gathering information from different sources, and also showed the areas where PROTEUS was able to provide a unique set of information about the experience of using a piece of software. PROTEUS proved reasonably reliable, and collected information with little need for monitoring by the evaluation researcher.

Drawbacks

Although PROTEUS is very easy to use to collect information, requiring little involvement by the researcher, it does have the drawback as it needs a number of prototypes. These are used as representations of the design space of a particular application interface. The main difficulty lies in having to implement several prototypes for a particular system, however it is anticipated that prototyping tools will assume a greater and greater role in system development. Larger and more complex systems require larger efforts to produce in low-level code, at the same time the applications can be quickly and easily simulated in prototyping tools. An examination of the Macintosh and Windows development environments shows that powerful fast applications can be developed in lower level programming languages such as C, but only with considerable expenditure of time by the designer/programmer. Applications such as HyperCard or Visual Basic allow less powerful simulation of the application to be developed very quickly. The increased availability of fast prototyping tools are making this less of problem. The *design* of an interface can be explored using a throw-it-away prototyping technique more akin to the role sketching plays in other design disciplines. These sketches allow fairly standard usability engineering to be employed, but more importantly allow users to gain an impression of what is possible within a particular design space. It also focuses a designer' mind on what users are likely to feel are important dimensions to describe that design space. The significant dimensions are where attention should be focused.

A second drawback of PROTEUS is the potential difficulty of interpreting output from the FOCUS algorithm. A major area of activity is the development of software tools to assist in the interpretation of FOCUS data, and to help in the comparison of data from different individuals. This will help draw out the comparison of designer and user models of the design space, and individual prototypes. A first step in this direction is the definition of guidelines for use by those inexperienced in FOCUS analysis.

Summary

Fitting PROTEUS into Usability Evaluation Frameworks

There are several ways of classifying usability evaluation methodologies, and PROTEUS can also be fitted into those classification schemes. Whitefield uses a classification schema to distinguish between usability methodologies based on the completeness of the interface Whitefield, 1990, Whitefield, et al., 1991.

	Users	
Computer Systems	Representational	Real
Representational	Analytical	User Reports
Real	Specialist Reports	Observations

Whitefields classification of usability methodologies.

This four class schema distinguishes between representations of users, and computer systems, and existing users and computer systems. Thus analytical techniques (such as Reisner's BNF, Reisner, 1981) fit into the upper left hand box of the grid. It is possible to perform a BNF evaluation without real users and without an implementation of the interface, only a specification is necessary. When the interface has been implemented, and is evaluated by experts, the method of evaluation (specialist evaluation) belongs in the lower left hand box of the grid. In this case the interface exists but the user is only represented in the expert's knowledge. User reports are where the user gives his opinions of a particular

interface, but the interface may only be represented by a non-functional prototype, or even by paper and pencil scenarios, equally the user will be reporting on recall of his experience with an interface, and therefore the interface can be considered partially representational. The final box of the grid is where the interface has been implemented as a functional prototype, and actual users are observed in their interactions with the prototype. Gould et al describe a system development in which these final three methods were all used in one form or another (Gould, et al., 1987). PROTEUS uses real users, and functional prototypes, assessment takes place during but not simultaneously with usage. This seems to place PROTEUS in the lower right hand box of the grid, but close to the border between real and representational systems.

Maguire identifies a number of issues in usability evaluation (Maguire and Sweeney, 1990):-

* Realism of Laboratory setting
* Ease of Learning versus Ease of Use
* Setting evaluation goals
* Feedback into the design process
* Need for user involvement
* Concurrent versus retrospective
* Degree of Intrusion
* Large amount of data
* Use of statistics

How does PROTEUS meet these issues? A considerable amount of care has been taken to allow the methodology to be used in natural settings (for example on users own machines). The nature of the tasks chosen for prototype systems has not always allowed the task setting to be completely natural. The task therefore assumes a slightly different (but not necessarily inappropriate) nature. The task becomes an exploration/familiarisation task, that a user who obtained a new piece of software might make.

Ease of learning and ease of use is an issue that can be looked at in several ways. The task for the user evaluator is fairly hard work, it requires some linguistic ability. Virtually all subjects were able to perform the task, suggesting that the task (though quite demanding) is fairly easy to learn. Post experiment comments suggest that users find the task quite hard work. The methodology is particularly easy to apply by the designer.

The analysis of the data from the experiment is partially automated, however currently analysis is completed as a qualitative abstracting task. The development of tools to help in this analysis remains a long term goal not fully explored at the moment. The feedback of results into the design process takes the role of highlighting those areas of a designer's knowledge that does not match with user knowledge. It is in these areas that designers will be least likely to produce design solutions that satisfy users, and this should form a focus for designer user discussion.

Usability Engineering, and the setting evaluation goals is not clearly supported by the PROTEUS methodology. However there is an implied goal that during development the interface should be more likely to be classified on the preferred side of evaluative constructs.

Users are centrally involved in the evaluation process, and during the process they become more aware of the possible alternatives that can exist. This partially addresses the problem of differing status between designers and users. The designer has defined a design space within which designs can be easily manipulated. The process of construing also forces users to define vaguely felt feelings into verbal descriptions, that in turn can ease later discussion between designers and users.

In the issue of concurrent versus retrospective evaluation, concurrent evaluation is usually regarded as more reliable than retrospective evaluation (for example Whiteside, et al., 1988). (Concurrent methods include verbal protocol collection, retrospective methods include survey

and interview data.). Concurrent collection of experience means that transient thoughts are more likely to be caught, and the record is likely to be accurate, recording real processes, rather than post-hoc justifications. The problems of concurrent methods of data collection are that they intrude into the process of completing a task, and so can change it dramatically. PROTEUS is in a fairly strong middle ground between concurrent and retrospective methods. Although the method is retrospective, the user can readily switch between the task under evaluation and the CES construing task. This means the time for thoughts to be lost is reduced, and yet intrusion onto the task is minimised. The PROTEUS methodology is best described contiguous rather than concurrent or retrospective. It is the contiguous nature of the evaluation that minimises intrusion on the evaluation task. The user/evaluator can complete tasks, or components of tasks, then leave the system for evaluation and start CES. Equally the user/evaluator can leave CES and enter the evaluation prototypes to check on some aspect. The process of construing frequently makes users aware of parts of the evaluation prototypes that they cannot quite remember, which they need to check. The rapid movement between evaluation prototype and CES supports accurate accounts on the interfaces without resulting in excessive interference with the task.

CES does produce large amounts of data, although this is immediately reducible via appropriate statistical methods. Cluster analysis is a method that has been frequently used with repertory grid data. It can be used to generate meaningful patterns from the mass of CES data. PROTEUS also generates system usage data. This data can be used in different ways depending on the task given to the user/evaluator. In an exploration task this type of data can indicate the time taken to become familiar with a particular interface. During construing it may indicate which interfaces are more easily remembered. In a longer, usage task the usage pattern can indicate preferences between the different prototypes.

Acknowledgements

The work described in this paper was conducted at the Open University as part of the author's Ph.D. thesis, supervised by Dr. J. Preece and Mr. D. Benyon. The evaluation of usability evaluation techniques was sponsored by Brameur as part of ESPRIT Project 1257, Muse.

References

J. H. Boose (1985) 'Personal construct psychology and the transfer of human expertise', in *Advances in artificial intelligence*, T. O'Shea [Ed(s)], Elsevier/North-Holland, Amsterdam.

S. K. Card, T. P. Moran and A. Newall (1980) 'The Keystroke-Level Model for User Performance Time with Interactive Systems' *Communications of the ACM,* 23, 7,

J. M. Crellin (1988) Personal Construct Psychology and the Development of a Tool for Formative Evaluation of Software Prototypes, *Proceedings of the Fourth European Conference on Cognitive Ergonomics*, Cambridge.

J. M. Crellin (1989) Personal Construct Psychology and Interface Design, Technical Report of the Faculty of Mathematics, Open University, No: 89., Milton Keynes.

J. M. Crellin (1990) PROTEUS: an approach to interface evaluation, *Proceedings of Interact '90*, North-Holland, Cambridge.

J. M. Crellin (1992) *Qualitative evaluation of software prototypes using repertory grid methods* unpub. Ph.D. thesis, The Open University.

J. M. Crellin, T. Horn and J. J. Preece (1990) Evaluating evaluation: an empirical examination of novel and conventional usability evaluation methods, *Proceedings of Interact '90*, North-Holland, Cambridge.

J. M. Crellin and J. J. Preece (1991) Case study of using different evaluation techniques in a small company *Information and Software Technology*, 33, 5, 366-382.

J. R. Davies (1990) 'A methodology for the design of computerised qualitative research tools' *Interacting with Computers* 2, 1, 33-57.

K. A. Ericsson and H. A. Simon (1984) *Protocol Analysis: Verbal Reports as Data*, The MIT Press, Cambridge, Mass.

W. A. K. Frost and R. L. Braine (1967) The application of the repertory grid technique to problems in market research *Commentry* 9, 3, 161-175.

J. Gould, S. Boies, S. Levy, J. Richards and J. Schoonard (1987) The 1984 Olympic Message System: A test of behavioural principles of system design *Communications of the ACM* 30, 9,

M. Graves (1976) 'The necessity for drawing: tangible sketching' *Architectural Design* 77, 6,

S. T. Gregory (1984) 'On prototypes versus mockups' *ACM SIGSOFT Software Engineering Notes* 9, 5, 13.

S. Hekmatpour and D. Ince (1988) *Software prototyping, formal methods and VDM*, Addison-Wesley, Wokingham.

R. Hewson (1990) *Sketching by Numbers: Is typography possible in the electronic paradigm?*, OU Technical Report 90/3, Walton Hall, Milton Keynes.

G. Kelly (1955) *A Theory of Personality: The Psychology of Personal Constructs*, Norton, New York.

R. J. Koubeck, G. Salvendy, R. Eberts and H. Dunsmore (1987) 'Eliciting knowledge for software development' *Behaviour and Information Technology* 6, 4, 427-440.

J. Long and A. Whitefield (1986) 'Evaluating Interactive Systems', in *PMT-607 Human-Computer Interaction, Unit 7, Evaluation*, The Open University Course Team [Ed(s)], The Open University, Milton Keynes, UK.

M. Maguire and M. Sweeney (1990) *Laboratory based methods and computer support tools for evaluating human computer interaction*, London.

A. Marcus (1983) 'Graphic Design for Computer Graphics', in *Readings in Human-Computer Interaction: A multidisciplinary approach*, R. M. a. B. Beacker W. A. S. [Ed(s)], Morgan Kaufmann Publishers, Inc., Los Altos, CA.

G. Martin, J. Pittman, K. Wittenburg, R. Cohen and T. Parish (1990) Sign here Please!, *Byte,* 15, 7.

M. J. Muller (1991) PICTIVE - An exploration in participative design, *Proceedings of CHI '91,* ACM,

J. Nielsen and R. Molich (1990) *Heuristic evaluation of user interfaces, Proceedings of CHI '90,* ACM, Seattle, WA.

D. A. Norman (1986) 'Cognitive Engineering', in *User centred system design: New perspectives on human-computer interaction*, D. A. Norman and S. W. Draper [Ed(s)], Lawrence Erlbaum, Hillsdale, N.J.

P. Reisner (1981) 'Formal Grammar and Human Factors Design of an interactive Graphics System' *IEE Transactions on Software Engineering* SE-7, 2, March, 229-240.

J. I. Robson and J. M. Crellin (1989) 'A model of the user's perceived control in interface design employing verbal protocol analysis' *Journal of Applied Ergonomics* 20, 4, 246-251.

M. L. G. Shaw (1978) 'Interactive computer programs for eliciting personal models of the world', in *Personal Construct Psychology 1977*, F. Fransella [Ed(s)], Academic Press, London.

M. L. G. Shaw (1980) *On Becoming a Personal Scientist*, Academic Press,

M. L. G. Shaw (1984) *PLANET User Manual*, Centre for Person-Computer Studies, Concord, Ontario, Canada.

V. Stewart and A. Stewart (1981) *Business Applications of Repertory Grid*, McGraw-Hill,

StudioBoxLtd. (1990) *Multi-Ad Creator*, Studio Box Ltd., Reading, Berks.

H. Thovtrup and J. Nielsen (1991) 'Assessing the Usability of a User Interface Standard', in *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, .

A. Whitefield (1990) *A framework to support human factors evaluation*, London.

A. Whitefield, F. Wilson and J. Dowell (1991) 'A framework for human factors evaluation' *Behaviour and Information Technology* 10, 1, 65-79.

J. Whiteside, J. Bennett and K. Holtzblatt (1988) 'Usability Engineering: Our experience and evolution', in *Handbook of Human Computer Interaction*, M. Helander [Ed(s)], Elsevier Sciences Publishers, Amsterdam.

Formal Methods in HCI: Moving Towards an Engineering Approach

Alan J. Dix*
H.C.I. Group, Department of Computer Science,
University of York, Heslington, United Kingdom, YO1 5DD.
E-mail: alan@minster.york.ac.uk
Tel: (0904) 432778

February 9, 1993

Abstract

The author and others have been studying the interplay of formal methods and HCI for several years. In particular, much of this work has centred on the design of formal models of interactive systems which can be used to formalise properties of usability. Although this work has been successful, it requires quite a high level of mathematical sophistication and is thus hard to 'give away' to the practitioner. This paper will describe two methods which have their roots in formal analysis, but which do not require great formal expertise. Such methods can be thought of as operating at an 'engineering level' as they have to some extent pre-packaged the results and insights of more sophisticated analysis into a form more readily applied to practical problems.

1 Introduction

For many years I have worked on the interplay between formal methods and human–computer interaction. This area of research is particularly associated with (present and past) workers from York and there are now several books on aspects of this area. Various books and papers are described in the annotated bibliography at the end of this paper. In particular, most of the material in this paper can be found in an expanded form in a HCI text book which I recently co-authored [1], and in my previous monograph [2].

Much of my work has concerned the development of formal models of interactive systems. Of these, the PIE model, described in Section 2, is one of the oldest and most well known. This model was designed to express generic properties at the level of WYSIWYG (what you see is what you get) or the meaning of undo. Such models can be used in two principle ways:

(i) They give new insight into general problems as the properties of the problem domain are analysed.

(ii) They can be used as part of a formal development process to constrain the design of specific systems.

The models have been successfully used on both counts. unfortunately, they require a considerable degree of expertise in both formal methods and human factors. If we assume that the use of formal methods in software engineering continues to gain favour and also that the importance of good interface design continues to be regarded as important, then it may be that such combinations of skills become more common. However, whether or not this comes to pass, such skills are not at present the norm.

Considering point (i) this is not a problem. An expert analyst can gain new insight from the models and then either expound this knowledge in the context of the models, or even recast

*Alan Dix is funded by a SERC Advanced Fellowship B/89/ITA/220.

it completely in informal terms. Indeed, this has been the pattern of much of my own work! The reader or listener need not have the same level of familiarity with the model to follow the arguments, even where the model is used in the exposition.

However, point (ii), the use in a formal development process, requires that such an expert be available on the design team. Furthermore, the burden of proof can become excessive. The real expert is able to focus on critical areas, but the proficient practitioner may become lost in the morass of detail. This problem is not unique to the use of formal methods in interface design, but is a general problem in all applications of formal methods within software engineering.

One could wait for research in software engineering to attack the general problem, possibly with the aid of complex proof assistants and toolsets. But, like waiting for the effects of education to filter through, this is at best a long term hope.

Happily, there are developments which make use of methods derived from these models, yet which require less formal expertise on the part of the designer. These I will call 'engineering' level techniques. In these, the theoretical work is effectively packaged in the same way that in civil engineering the theoretical analysis of materials and soil mechanics is packaged up into tables, structural analysis programs and rules of thumb.

I will describe two such 'engineering' level approaches. The first, in Section 3, shows how properties similar to those defined over formal interaction models can be applied to dialogue descriptions. This has the advantage that dialogue descriptions are often produced for specification or prototyping purposes and that the analyses are simpler and can often be automated.

The second approach is status/event analysis This uses a distinction drawn from formal modelling work, but which can give insight into general and specific interface design with little reference to its formal roots. In particular, I will demonstrate how status/event diagrams can be used to analyse the usability of interface widgets.

2 Formal models of interaction

It's easy to say that a system is *WYSIWYG*, is *consistent* or has a universal *undo* facility, but how do you know? Is there any way we can take a system and say 'yes' it is or 'no' it is not WYSIWYG? It was just for this purpose that formal models of interaction were developed. One of the oldest such models is the *PIE* [5]. This is a general purpose model intended to apply to a wide range of systems. More specific models can address particular design areas, for example, there are models for windowing systems, models for describing temporal behaviour and models for dealing with non-determinism[2]. At the opposite extreme from the PIE model are specifications of specific systems, for example, Sufrin's specification of a text editor [6]. These of course can only be used to reason about the one system (although the specifier may draw general lessons).

The rest of this section will concentrate on the PIE model and a few of the usability properties which it can be used to describe. We will look at two broad classes of properties. Those concerning what you can *see* of the system: predictability and observability, and those concerning what you can *do* to the system: reachability and undo. The concept of WYSIWYG is partly captured within the PIE's *predictability* property, but like all usability properties, we find that the formal statement is only a partial expression of the real property we desire. Nevertheless, some of the formal principles are necessary for usability, any system which breaks them is bound to have problems. The formal principles form a 'safety net' to prevent some of the worst mistakes in an interactive system. Formal analysis is thus an aid to, but *not* a substitute for, good design.

Concepts underlying the PIE model

The PIE model is a *black-box model.* It does not try to represent the internal architecture and construction of a computer system, but instead describes it purely in terms of its inputs from the user and outputs to the user. For a simple single-user system typical inputs would be from keyboard and mouse, and outputs would be the computer's display screen and the eventual printed output (Figure 1).

The difference between the ephemeral *display* of a system, and the permanent *result* is central to the PIE model. We will call the set of possible displays D and the set of possible results R. In order to express principles of observability, we will want to talk about the relation between display

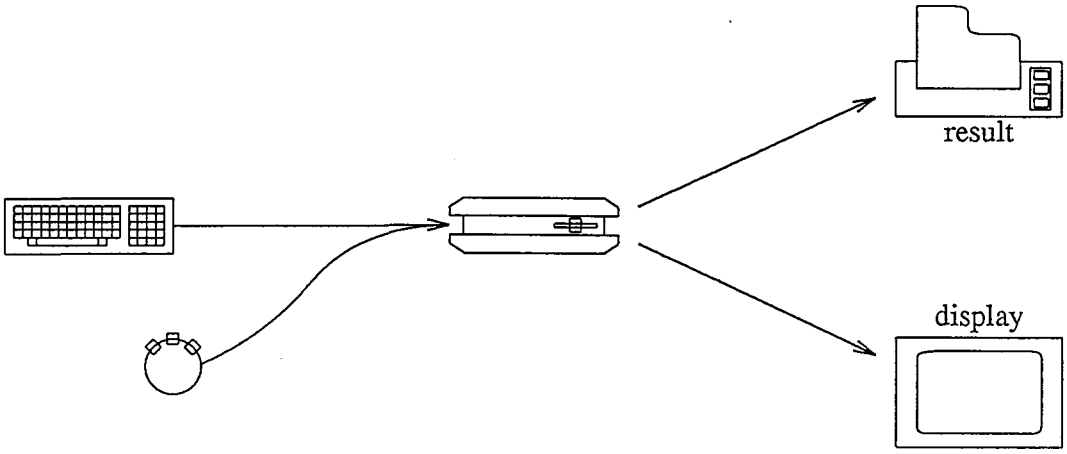


Figure 1: Inputs and outputs of single user system

and result. Basically, can we determine the result (what you will get) from the display (what you see)?

More formally ...

For a formal statement of predictability it helps (but is not essential) to talk about the internal state of the system. This does not counter our claim to have a black-box model. First, the state we define will be opaque, we will not look at its structure, merely postulate it is there. Second, the state we will be discussing is not the actual state of the system, but an idealisation of it. It will be the minimal state required to account for the future *external behaviour*. We will call this the *effect* (E). Functions *display* and *result* obtain the current outputs from this minimal state:

$$\mathit{display} : E \rightarrow D$$
$$\mathit{result} : E \rightarrow R$$

The current display would be literally what is now visible. The current result is actually not what *is* available, but what the result would be if the interaction were finished. For example, with a word-processor it is the pages that would be obtained if one printed the current state of the document.

A single user action we will call a *command* (from a set C). The history of all the user's commands is called the *program* ($P = \mathrm{seq}\, C$), and the current effect can be calculated from this history using and *interpretation function*:

$$I : P \rightarrow E$$

Arguably the input history would be better labeled H, but then the PIE model would loose its acronym!

If we put together all the bits, we obtain a diagram of sets and functions (Figure 2), which looks rather like the original illustration.

In principle, one can express all the properties one wants in terms of the interpretation function I. However, this often means expressing properties quantified over all possible past histories. To make some of the properties easier to express, we will also use a state transition function *doit*:

$$\mathit{doit} : E \times P \rightarrow E$$

The function *doit* takes the present state e and some user commands p, and gives the new state after the user has entered the commands $\mathit{doit}(e, p)$. It is related to the interpretation function I by the following:

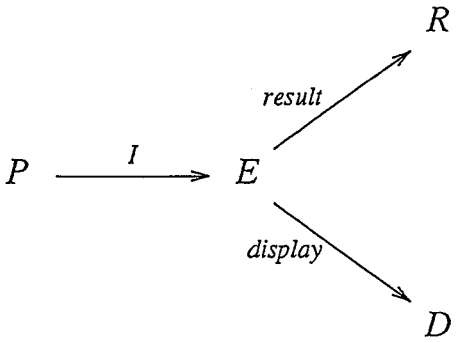


Figure 2: The PIE model

$$
\begin{aligned}
doit(I(p), q) &= I(p \frown q) \\
doit(doit(e, p), q) &= doit(e, p \frown q)
\end{aligned}
$$

Now the PIE diagram can be read at different levels of abstraction. On can take a direct analogy with Figure 1. The commands set C is the keystrokes and mouse clicks, the display set D is the physical display, and the result R is the printed output.

$$
\begin{aligned}
C &= \{\text{'a'}, \text{'b'}, \ldots, \text{'0'}, \text{'1'}, \ldots, \text{'*'}, \text{'\&'}, \ldots\} \\
D &= Pixel\_coord \rightarrow RGB\_value \\
R &= \text{ink on paper}
\end{aligned}
$$

This is a physical/lexical level of interpretation. One can produce a similar mapping for any system, in terms of the raw physical inputs and outputs. It is often more useful to apply the model at the logical level. Here the user commands are higher level actions such as 'select bold font' which may be invoked by several keystrokes and/or mouse actions. Similarly, we can describe the screen at a logical level in terms of windows, buttons, fields etc. Also, for some purposes, rather than dealing with the final physical result, we may regard say the document on disk as the result.

The power of the PIE model is that it can be applied at many levels of abstraction. Some properties may only be valid at one level, but many should be true at all levels of system description. It is even possible, to apply the PIE model just *within* the user, in the sense that the commands are the user's intended actions and the display, the perceived response.

When applying the PIE model at different levels it is possible to map between the levels. This leads to *level conformance* properties, which say, for example, that the changes one sees at the interface level should correspond to similar changes at the level of application objects.

Observability and predictability

The WYSIWYG is clearly related to what can be inferred from the display (what you see). Harold Thimbleby has pointed out that WYSIWYG can be given two interpretations [4]. One is what you see is what you *will get* at the printer. This corresponds to how well you can determine the result from the display. The second interpretation is what you see is what you *have got* in the system. For this we will ask what the display can tell us about the effect. These can both be thought of as *observability* principles.

A related issue is *predictability*. Imagine you have been using a drawing package and in the middle you get thirsty and go to get a cup of tea. On returning, you are faced with the screen – do you know what to do next. If there are two shapes one on top of the other, the graphics package may interpret mouse clicks as operating on the 'top' shape. However, there may be no visual indication of which is topmost. That is the screen image does not tell you what the effect

of your actions will be, you need to remember how you got there, your command history. This has been called the '*gone away for a cup of tea problem*'.

In fact the state of the system determines the effects of any future commands, so if we have a system which is observable in the sense that the display determines the state, it is also predictable. Predictability is a special case of observability.

Let's have a go at formalising these properties. To say that we can determine the result from the display is to say that there exists a function $transparent_R$ from displays to results.

$$\exists\, transparent_R : D \rightarrow R \\ \bullet\ \forall\, e \in E \bullet transparent_R(display(e)) = result(e)$$

Of course, there is no good having any old function from the display to the result, the second half of the above says that the function gives us exactly the result we would get from the system. This property is a good first cut at *observability*, but will in fact turn out to be too strong. For now call it *result transparency*. It says that the display contains at least as much information as the result. However, it may also contain additional information about the interactive state of the system. For example, you will observe the current cursor position, but this has no bearing on the printed document.

So you know what will happen if you hit the print button *now*. Refreshed from your cup of tea, you return to work. You press a function key which, unbeknown to you, is bound to a macro intended for an entirely different application. The screen rolls, the disk whirs and to your horror your document and the entire disk contents are trashed. You leave the computer and go for another drink ...not of tea.

A stronger condition can be obtained if we demand that the system state can be observed from the display:

$$\exists\, transparent_E : D \rightarrow E \\ \bullet\ \forall\, e \in E \bullet transparent_E(display(e)) = e$$

We can regard this as an initial attempt at *predictability*, but, it will again turn out to be too strong, so instead we will call the above property simply *transparency*.

What would it mean for a system to be transparent in one of these senses. Well, if the system were result transparent, when we come back from our cup of tea, we can look at the display and then work out in our head (using $transparent_R$) exactly what the printed drawing would look like. Of course, whether we could do this in our heads is another matter. For most drawing packages the function would be to simply ignore the menus and 'photocopy' the screen.

Simple transparency is stronger still. It would say that there is nothing in the state of the system that can not be inferred from the display. If there are any modes, then these must have a visual indication, if there are any differences in behaviour between the displayed shapes, then there must be some corresponding visual difference. Even forgetting the formal principles, this is a strong and useful design heuristic.

Unfortunately, these principles are both rather too strong. If we imagine a word-processor rather than a drawing package, the contents of the display will be only a bit of the document. Clearly, we cannot infer the contents of the rest of the document (and hence the printed result) from the display. Similarly, to give a visual indication of say object grouping within a complex drawing package might be impossible (and this can cause the user problems).

On the other hand, one could regard the transparency properties as being too *weak*. The function $transparent_R$ represents the reasoning the user would have to do to predict the result from the current display. That such a function exists is no guarantee that the user can perform the calculation – it may involve the translation of Ancient Egyptian hieroglyphics.[1]

When faced with a document on a word-processor, the user can simply scroll the display up and down to find out what is there. That is, you cannot see from the current display everything about the system, but you can find out. The process by which the user explores the current state of the system is called a *strategy*. The formalisation of a strategy is quite complex, even ignoring cognitive limitations. These strategies will differ from user to user, but the documentation of a system should tell the user how to get at pertinent information. For example, how to tell what

[1] Which is precisely how non-English speakers feel when faced with many menu driven interfaces.

objects in the drawing tools are grouped. This will map out a set of effective strategies with which the user can work.

One can use the idea of a strategy to formulate more effective observability and predictability properties. Indeed, one can go further still and generate models and properties which take into account aspects of user attention, and issues like keyboard buffers. The books referred to in the bibliography deal with such extensions.

Principles of predictability do not stand on their own even if you had known what was bound to the function key, you might still have hit it by accident, or simply forgotten. Other protective principles like *commensurate effort* need to be applied [4]. Also, although it is difficult to formalise completely, one prefers a system which behaves in most respects like the transparency principles, rather than requiring complicated searching to discover information. That is a sort of commensurate effort for observation.

Reachability and undo

In a commercial program debugger, there is a window listing all the variables. If a variable is a complex structure, then hitting the 'insert' key while the cursor is over the variable will expand the variable showing all its fields. If you only want a few of the fields to be displayed, you can move the cursor over the unwanted fields and type the 'delete' key and the field is removed. These operations can be repeated over complex hierarchical structures. If you remove a field and then wish you hadn't, you can always press insert' again over the main variable and all the fields will be re-displayed. Even this breaks somewhat the principle of *commensurate* effort, but worse is to come. The 'delete' key also works for top level variables, but once one of these is removed from the display there is *nothing* you can do to get it back, short of exiting the debugger and re-running it from scratch.

A principle which stops this type of behaviour is *reachability*. A system is reachable if from any state the system is in, you can get to any other state. The formal statement of this is as follows:

$$\forall e, e' \in E \bullet \exists p \in P \bullet doit(e, p) = e'$$

Unlike the predictability principles, there are no awkward caveats. The only problem is that, if anything, it is too weak. For instance, a word-processor could have a delete key, but no way to move the cursor about, you always type at the end of the document. Now you can, of course, get from any document state to any other, you simply delete the whole text and retype what you want. However, if you had just typed in a whole letter then noticed a mistake on the first line, you would not be pleased! So, ideally one wants an independent idea of 'distance' between states and make the difficulty of the path between them commensurate with the distance – small changes should be easy. Despite this, the principle on its own would have been strong enough to prevent the behaviour of the debugger!

One special case of reachability is when the state you want to get to is the one you have just been in. That is, *undo*. We expect undo to be easy, and ideally have a single undo button that will always undo the effect of the last command. We can state this requirement very easily:

$$\forall c \in C \bullet doit(e, c \frown undo) = e$$

This says exactly what we wanted. We start in a state e. We then do any command c and follow it by the special command *undo*. The state is then the same as we began in.

Stop! Before clapping ourselves on the back for so clearly defining undo, we should check that this requirement for undo is consistent. Indeed, it is consistent – so long as there are at most two states. That is, the above undo requirement is only possible for systems which *do* virtually nothing! The reason for this is that *undo* is itself a command and can undo itself. Take any state e and choose any command x. Let e_x be the state you get to after command x. That is $e_x = doit(e, x)$. Now we can apply the undo requirement to state e_x:

$$doit(e_x, undo) = doit(e, x \frown undo) = e$$

So, the *undo* command in state e_x gets us back to e. That is as expected. But what does *undo* do if we are in state e. Again we can employ the undo principle remembering that $e = doit(e_x, undo)$:

$$doit(e, undo) = doit(e_x, undo \frown undo) = e_x$$

This uses the undo principle when the command c is undo itself. However, our choice of command x was arbitrary, so if we had chosen another command say y we would have concluded that $doit(e, undo) = e_y$. This means that $e_x = e_y$, and in general anything we do from state e gets us to the same state. With a little more argument you can show that any command from this second state gets us back to the original one. So at very most we have two states, a toggle, with all the commands flipping back and forth between them. The only other alternative is that the system does nothing.

We won't go on to describe the details of better undo requirements, the interested reader can find that elsewhere. The basis of most workable undo systems is that *undo* is not just any old command, but is treated differently. The simplest fix to the above undo principle is to restrict the commands to anything *except undo*!

The lesson from the above is clear. It is easy to say you want something which sounds quite reasonable. A formal description of the requirement may well reveal that, as in the case of undo, it is inconsistent – that is *no* system could be built which satisfies the requirement.

Summary – formal modelling

The example of undo shows how useful formal models can be as tools for understanding. The specification we originally gave sounded good enough, but was inconsistent. If we had tried to build a system having such an undo, we would either fail, or *think* we had succeeded. In the former case, we might keep fruitlessly trying to build a system with a single universally applicable undo button. In the latter, we might delude ourselves into thinking this was what we had, only to discover (after selling the system!) that there were cases where it failed.

However, it is also clear that the job of verifying that a large interactive system is reachable or predictable may be very difficult.

3 Dialogue Analysis

The difficulty about proving properties of systems is that the state is very complex. For example, the state of a word processor will contain information such as:

Screen:	edit screen
Text:	"to be or not to be, ..."
Menu:	file menu displayed
Cursor:	at the 7th character line 12

To be able to prove things about such a state, we need to reason about numbers and text as well as mode indicators such as **Screen** and **Menu**. The number of possible texts and cursor positions is infinite, or even if we take into account system limits *very large*. This means we have to reason symbolically – heavy mathematics!

Dialogue descriptions usually limit themselves to the finite attributes of the state. Those which have a major effect on the allowable sequences of user actions. They are thus instantly more amenable to automated analysis (we can sometimes simply try all cases). Furthermore, dialogue descriptions are often used as part of design anyway, thus we may be able to take an existing product of the design process and obtain instant added value.

3.1 Notations

There are a large number of different dialogue notations. Some use diagrammatic representations of the dialogue (see below) and others use textual representations (such as the use of grammars or production rules).

Of the diagrammatic techniques, *state transition network* (STNs) are most heavily used. (But even they come in several variants.) We will base our discussion primarily on STNs, but other notations could equally be used.

State transition nets consist of two elements:

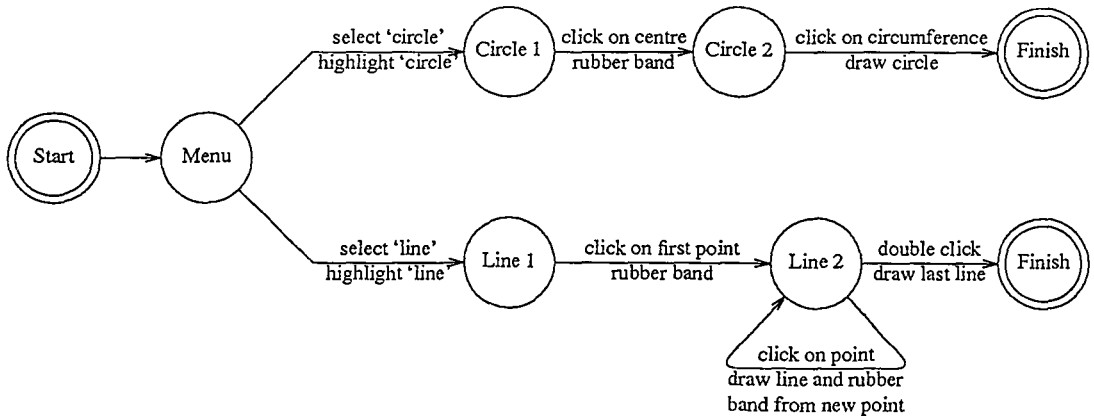


Figure 3: State transition network for menu driven drawing tool

circles – denoting the states of the dialogue

arcs – between the circles, denoting the user actions/events

Figure 3 shows a STN describing a portion of the dialogue of a simple drawing tool. The arcs are also labelled with the feedback or system response resulting from the user's actions. Note how cramped the arcs get – obviously a lot is happening at each event.

The STN for a full system would usually be enormous. To manage the complexity, STNs are often described hierarchically. For example, Figure 4 shows the higher level dialogue for the drawing tool, selecting between several sub-menus. The menu in Figure 3 corresponds to the graphics sub-menu. Each of the sub-menus would have similar STNs describing them.

The hierarchical decomposition in this diagram is of states. Single states in the high-level diagram correspond to an entire low-level STN. There are other possibilities for hierarchical decomposition, for example, augmented transition networks allow both user actions and system responses to be decomposed into further STNs.

3.2 Why do people use dialogue notations?

I said that we were focusing on dialogue descriptions because they often 'come for free', a natural product of the design process. There are several reasons for this:

- The use of UIMS or UIDEs.
- For dialogue specification on paper.
- For rapid prototyping.

We'll look at these in turn.

UIMS

If we use a *User Interface Management System* (UIMS) or User Interface Development Environment (UIDE) this will usually include a formal description of the dialogue. This may be in the form of production rules, a grammar or even some graphical representation. Some of these representations, especially production rules, do not completely separate the dialogue from the underlying state. However, the conversion required is certainly far less work than generating the description from scratch *and* is guaranteed to be consistent with the actual system.

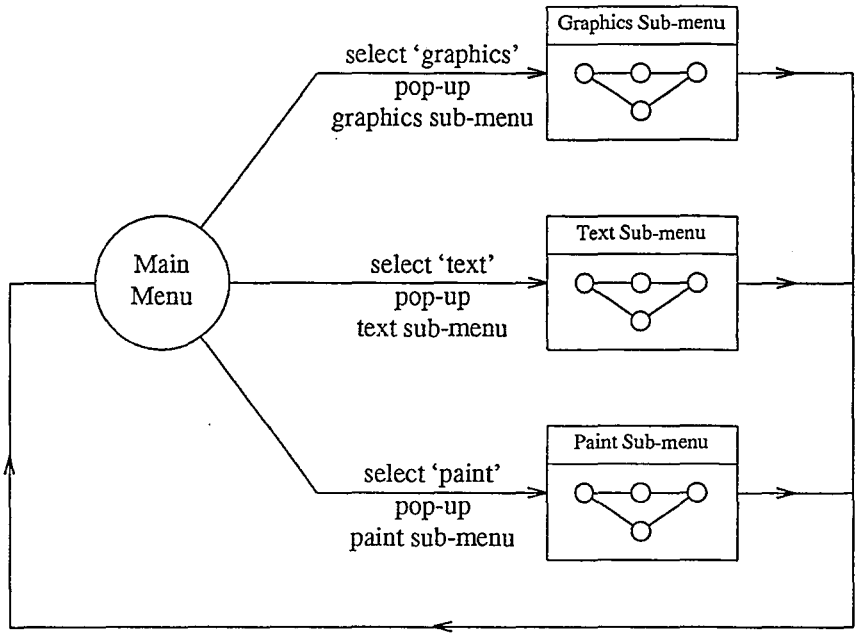


Figure 4: Hierarchical state transition network for complete drawing tool

Paper specification

A second reason for the use of dialogue descriptions is simply as a paper specification method, just as one might use data-flow diagrams for information systems or entity-relationship diagrams for database design. Several years ago I was working in a data processing department producing information systems under a forms-based transaction processing (TP) environment.

Programming a TP system is similar to many window systems, basically a stimulus-response model. Your program gets a screen full of data and must decide what to do with it. When it has processed that screen, it sends a fresh template to the user and then goes on to service a *different* terminal. Because of this form of programming, one cannot implicitly encode the dialogue within the program structure. So, for example, it is quite difficult to ensure that the user can only delete a record after it has been displayed.

To ease the problem of writing (relatively) complex dialogues under this regime, the author used flowcharts to describe the interaction with each user. Figure 5 shows a flowchart for a delete sub-dialogue similar to those used.

Note two things, despite surface similarities, there are important differences both from normal program flowcharts and from STNs.

First, note that a flowchart of the program implementing this dialogue would (because of the stimulus-response model) be tree-like. It would have to explicitly store the dialogue state and generally being totally incomprehensible *without* the corresponding dialogue description. Furthermore, the sorts of things one puts in the boxes of a dialogue flowchart are different from program flowcharts. For example, reading a record could be a complex activity, say searching through a file until the matching record is found. However, from the dialogue viewpoint this corresponds to a single system action.

Note also that although superficially like an STN, with boxes connected by arrows, the emphasis is rather different. The boxes represent system processes or user interactions, that is, the notation is event/process oriented rather than state/oriented. We will return to this status/event distinction in Section 4.

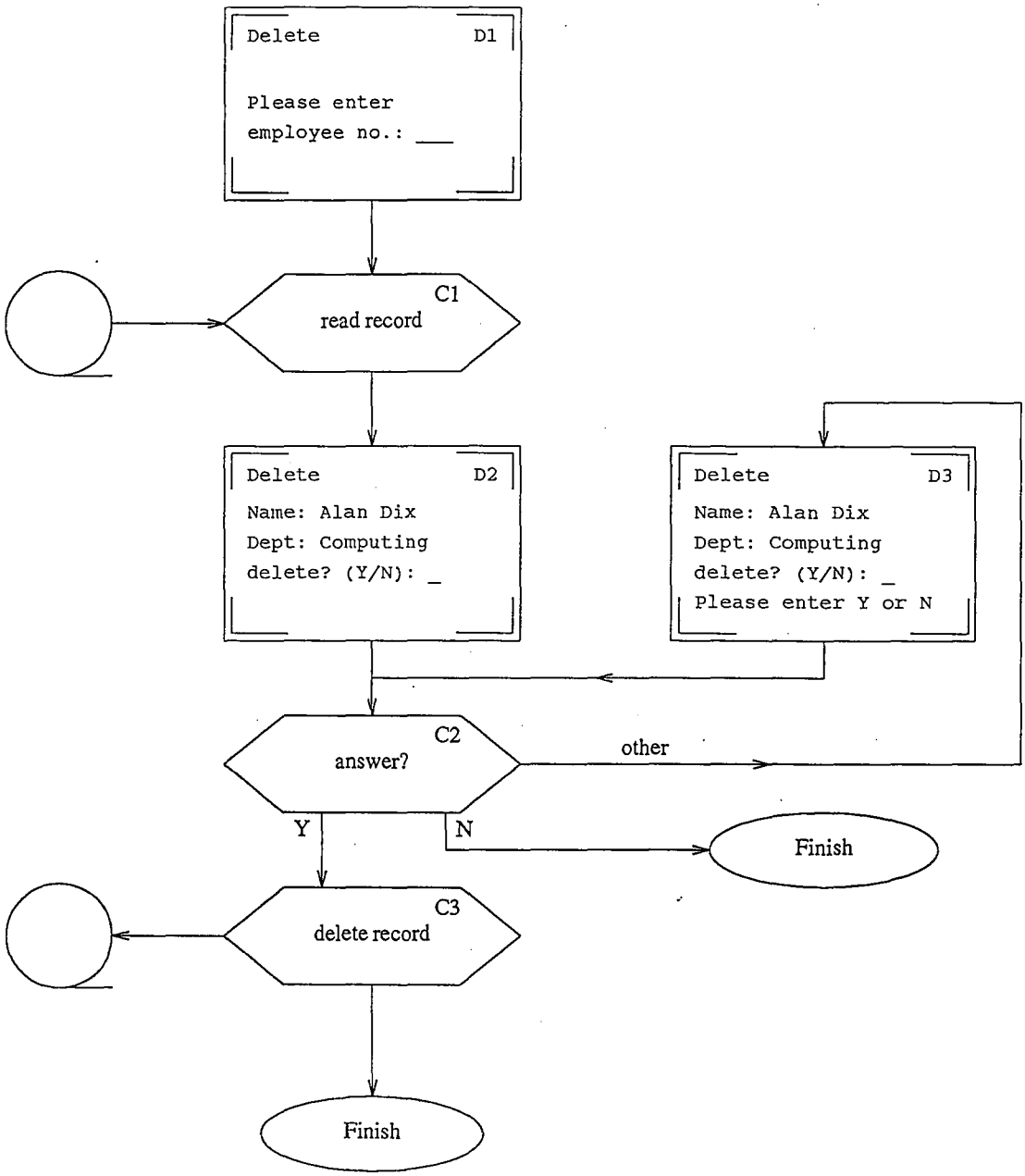


Figure 5: Flow chart of deletion sub-dialogue

In a different vein, formal notations are often criticised for the amount of work required. However, the author's experience counters this. The author used these diagrams and converted them, mechanically, but by hand into Cobol programs. Using this method I was able to produce within days, systems which had previously taken months to complete. Furthermore, changes could be accomplished within hours (no mean feat within such an environment!). Although, it might be nice to think this was due to superior programming skills (!), this could in no way account for an order of magnitude difference in productivity. That is, the adoption of a kind of formal notation did not waste valuable time, but instead made phenomenal time savings.

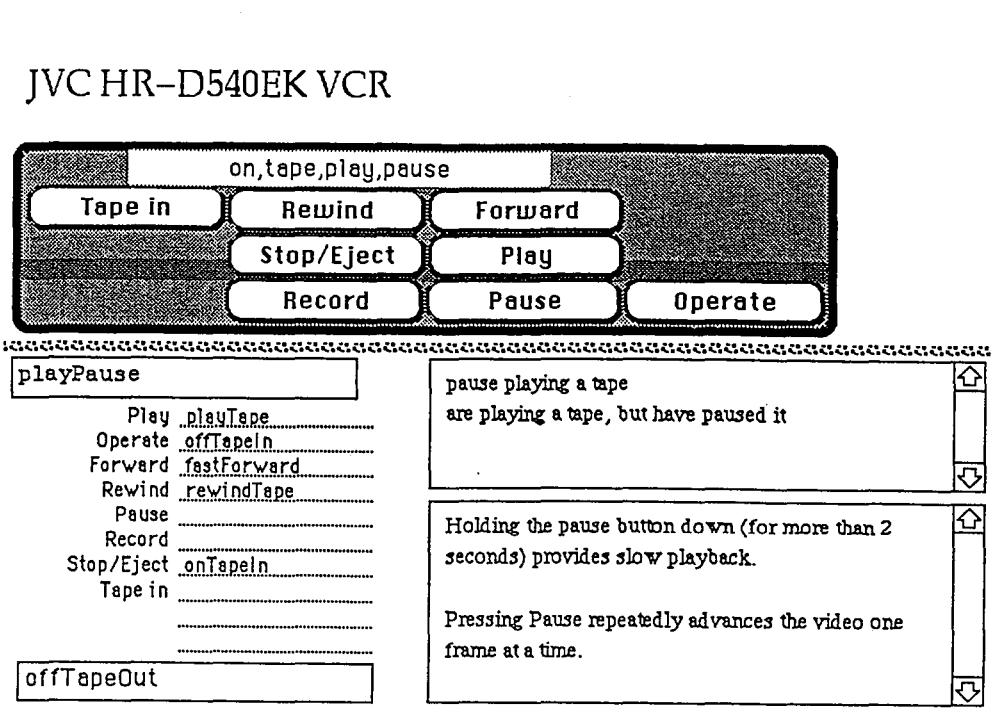


Figure 6: Hyperdoc

Prototyping

Dialogue descriptions can be used to drive prototyping tools or simulators. This is rather like the use with UIMS, but usually with a less extensive environment. One example of this is Heather Alexander's SPI notation (*Specification Prototyping and Interaction*) [11]. This uses a variant of CSP for the dialogue description and then has tools which allow one to 'run' the dialogue seeing the possible interaction paths.

Another support tool is Hyperdoc developed by Harold Thimbleby [12], shown in Figure 6. The screen shows part of the description for a JVC video-recorder. The top half of the screen is a drawing of the interface. The buttons on the drawing are active – the simulation runs when they are pressed. On the bottom left, we can see part of the dialogue description. This describes the transitions from the state 'playPause'. For example, if the user presses the 'Operate' button, the state will change to 'offTapein'.

In fact, this tool does more than simply simulate the dialogue, it can perform several forms of dialogue analysis.

3.3 Dialogue properties

Given a dialogue description, we can begin to look at what properties it satisfies. We look at these under two headings

action properties which describe local phenomena at a particular state. That is, concerning single actions.

state properties which concern the movement between states, which may encompass whole trains of actions.

After looking at these two kinds of properties, we will consider two examples of their use.

Action properties

There are several dialogue properties which are to do with local dialogue actions:

completeness – look at each state, is there an arc coming from that state for each possible user action? If not, what is the effect on the system if the user performs this action? This is a good way of checking for 'unforeseen circumstances'.

determinism – is the behaviour uniquely defined for each user action. In a simple STN this corresponds to checking that there is at most one arc labelled with each user action from a particular state. Non-determinism can be deliberate, corresponding to an application decision. However, it can be a mistake, and this is especially easy in complex hierarchical STNs, production rules systems etc. Automatic tools can help check for this.

consistency – does the same user action have a similar effect in different states? If not are these dialogue *modes* visibly different?

If we look back to Figure 3 we can check it for completeness. The action 'select-line' is not mentioned in either of the line states, but this is deliberate. The line option is assumed to be on a pop-up menu and so cannot occur except from the menu state. The remaining actions are then single and double clicks. What happens if we double click in either of the circle states? Is this signaled to the user as an error by a beep, simply ignored, does it do something odd (a feature!) or does it crash the program?

State properties

Another set of properties are more global, considering how easy or difficult it is to get from one state to another:

reachability – can you get anywhere from anywhere? That is, imagine you are at a particular dialogue state and you want to get to a different state. Is there a sequence of user actions which is guaranteed to get you there? In addition, we may want to ask just how complicated and long that sequence is.

reversibility – can you get to the previous state? Imagine you have just done an action, but wished you hadn't. This is a special case of reachability, but one which we expect to be especially easy – we all make mistakes. Note this is *not* undo – returning to a previous dialogue state does not in general reverse the semantic effect.

dangerous states – there are some states you don't want to get to. Does the system make it difficult to perform actions which take you to these dangerous states?

As an example, we can check the reversibility of the drawing tool (Figures 3 and 4). Imagine we want to reverse the effect of "select 'line'" from the graphics Menu state. We can perform three actions:

click – double click – select 'graphics'

These return us to the graphics pop-up menu. However, these will leave a vestigial circle on the display. That is, in this case, as we warned, reversing the dialogue is *not* undo.

Note also that this reachability for dialogue states is equivalent to the definition for full system states, but weaker. A system cannot be reachable in the PIE sense if it is not reachable at the dialogue level, but, like undo, dialogue reachability does not guarantee full reachability.

In graph theoretic terms, dialogue reachability is called *strong connectivity* and the Hyperdoc tool, described previously, is able to perform this analysis for the designer.

3.4 Example – Digital watch

3.4.1 User's documentation

A digital watch has a very limited interface – 3 buttons. These must control the watch display (time/calendar) a stopwatch mode and an alarm.

We only consider one of the buttons, button 'A', which is used to move between the four main modes: time/calendar, stopwatch, alarm setting and time setting.

Figure 7 shows a portion of the user instructions. It is a simple state transition network.

We can analyse this network. The time and alarm setting modes are dangerous states, we don't want to set the time by accident. These states are guarded – you have to hold the button down for two seconds. This button is very small and it is difficult to hold it down by accident.

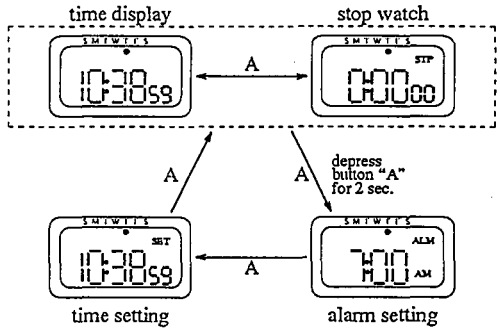


Figure 7: Instructions for digital watch

What about completeness? The idea of holding the button down suggests that we ought to distinguish the actions of depressing and releasing button 'A'. So, what do these actions do in the different modes?

Although the STN is incomplete this is acceptable for the user instructions so long as undocumented sequences of actions do not have a disastrous effect. However, the designer must investigate all possibilities to check this.

3.4.2 Designer's documentation

Extensive experiment eventually revealed the complete STN for the watch, shown in Figure 8. This includes for each state the effect of the three actions:

- depress A
- release A
- wait two seconds

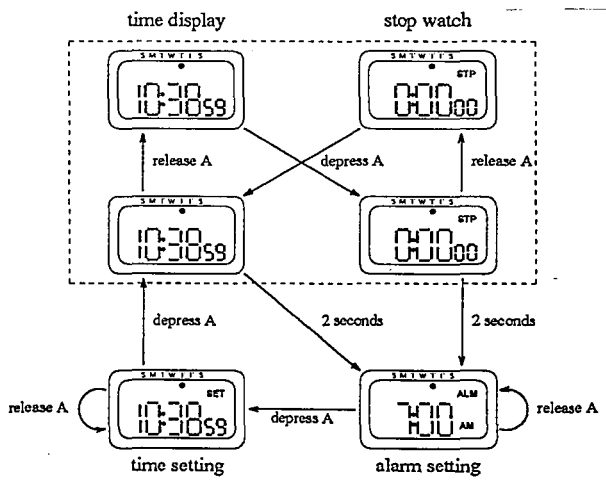


Figure 8: Design diagram for digital watch

Notice that this required the addition of two meta-stable versions of the time/calendar state and the stopwatch state. This is the sort of diagram that the designer would need to analyse and to pass on to the implementor.

The diagram looks fairly complex — and we've only looked at one button!

3.5 Example – Dangerous states

One of the word processors being used to prepare this document exhibits dangerous states. It has two main modes, the main mode where you edit the text, a menu and help screen from where you perform filing operations. You switch between these modes with the 'F1' key. In addition, from the menu you can exit the word processor by hitting the 'F2' key. These modes and the exit are shown in Figure 9.

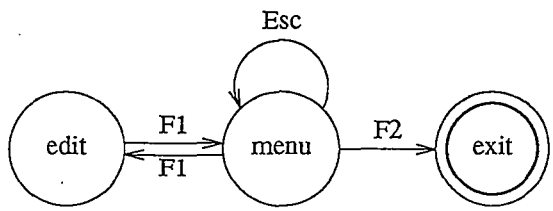


Figure 9: Main modes of text editor

If the text has been altered it is automatically saved upon exit. However, if you have altered the text, but then decide to abandon your edits, this automatic save can be turned off by hitting the escape key in the Menu mode. Subsequent edits will reset this and the text will be again be saved. Of course, not saving altered text is dangerous (but may be required). We therefore get the diagram in Figure 10 with dangerous states hatched.

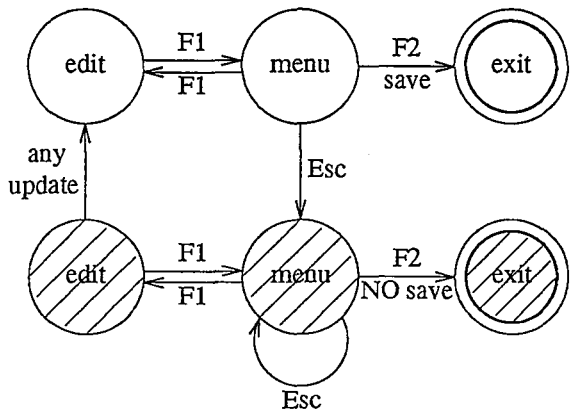


Figure 10: Revised STN with dangerous states

This multiplying of states is a *semantic* distinction, but can be recorded in the dialogue. We can then ask at a dialogue level whether or not it is easy to get into the dangerous states by accident. The user spends most of the time in the edit state, so the most dangerous sequence is 'F1-Esc-F2 – exit with *no* save. This is rather close to the sequence 'F1-F2' – exit *with* save, but is this mistake easy to make?

If we decided it was, we can insert a guard, such as a dialogue box asking for confirmation. In fact, the word processor has no such guard.

The dialogue is *not* as is sometimes claimed independent of presentation. There are various lexical and presentation issues which impinge on the dialogue. In particular, the layout of keys on a keyboard or menu items on a screen affects the sort of lexical errors which occur. For example, the author's old computer had the function keys on a separate keypad. One could not accidentally

hit 'Esc' in the middle of the sequence 'F1-F2'. However, the author's current keyboard layout is as in Figure 11 – disaster!

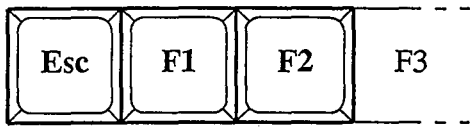


Figure 11: Dangerous function key layout

4 Status/event analysis

We have already seen the state/event dichotomy when discussing dialogue notations. Some are better at states others at events. Really both are needed to understand interactive systems.

Status/event analysis is a semi-formal, "engineering" level, technique which looks at the interplay between status and events in an interface [13, 1]. Status here refers to not just dialogue or system state, but any persistent aspect of the system, such as the display.

Status/event analysis is based on two foundations:

formal modelling – understanding gleaned from variants of the PIE model.

naïve psychology – common sense or very simple perceptual psychology.

It is 'engineering level' in the sense that you do not require a deep understanding of either formal models or psychology.

4.1 Properties of events

Consider alarm clocks. These demonstrate most of the important properties of status and events.

status: the watch face. You can look at the watch face whenever you like.

event: an alarm. It happens at a particular moment, if you aren't there when the alarm goes, you miss it.

status change event: when a particular time passes that is an event, whether or not anyone notices. In general, any change of status can be viewed as an event.

actual and perceived events: you don't necessarily notice an event straight away. There is usually a gap.

polling: if you are waiting for a particular time, you occasionally glance at the watch face. That is you poll the status to make the status change become a perceived event.

granularity: what constitutes an event depends on timescale. If we are interested in planning a day (timescale of hours) then an alarm is an event. However, whilst waking up (timescale of seconds or minutes) the alarm ringing is a status.

4.2 Design implications

Applications have timescales depending on the tasks which they fulfill. Events happen in the system and at the interface. Do these actual events become perceived events for the user, and if so, is the lag within an acceptable timescale for the application?

There are problems if perceived feedback is too slow or too fast.

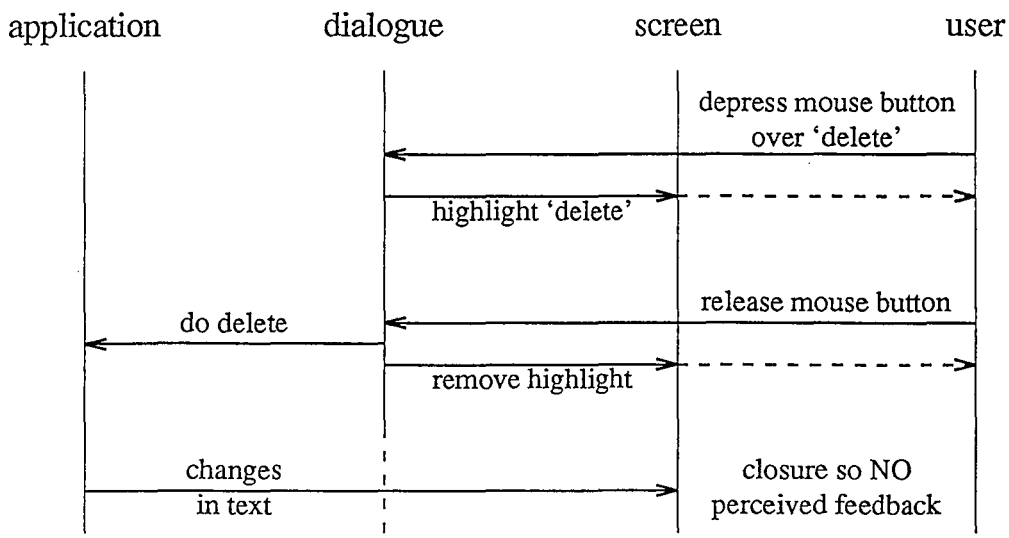


Figure 12: Screen button – hit

too slow – if the lag between actual and perceived event is too great, the recipient may respond too late. For example, if the operator in a power plant were sent an email saying 'meltdown imminent', the plant may have vapourised before the operator read the mail.

too fast – if the perceived event is too fast, it may interrupt a more immediate task. For example, if alarms began to ring when the stock of 6mm bolts was running low, then the operator may get distracted from dealing with the potential meltdown.

4.3 Naïve psychology

We can use simple psychology:

- We can predict where the user is looking:

  at the mouse – when positioning

  at the insertion point – intermittently when typing

  at the screen – if you're lucky

- We know certain effects cause immediate events:

  audible bell – when the user is in the room.

  peripheral vision – movement or large change.

- Closure at the end of a task has predictable effects:

  loss of attention – including the mouse pointer

  concurrent activity – the user may begin a new activity whilst finishing off the last one.

4.4 Example – screen button widget

On-screen buttons invoke an application action. They are activated by clicking the mouse over them However, users often mis-click the button. Furthermore the mistake is often missed. This is a common widget in virtually every graphical interface, and the error is common too. Why?

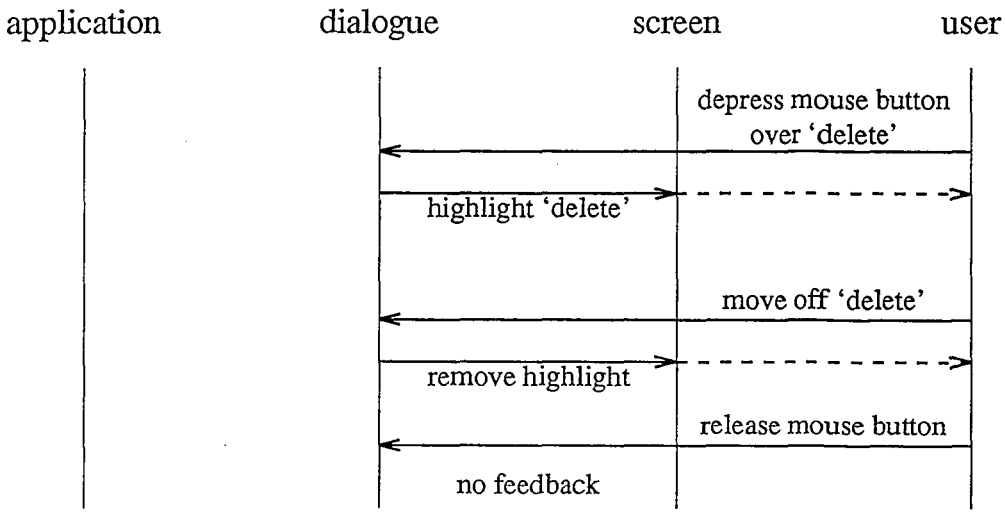


Figure 13: Screen button – miss

To analyse this problem we draw status-event diagrams (Figures 12 and 13). To do this we draw a timeline for each layer of the system: application, dialogue, screen and user. Time runs down the page and arrows between the timelines represent events.

Figure 12 shows the case when the button is successfully activated (a hit) and Figure 13 shows the case when the user slips the mouse off the screen-button just before the mouse-button is released (a miss).

The two diagrams are very similar. They differ in which user action occurs first, the release or the movement of the mouse. However, after the user has positioned the mouse over the target, closure is reached. Therefore the two actions occur concurrently. The mistake is likely.

Why is the mistake not noticed? The highlighting of the button is salient as it is at the target of a mouse positioning task. However, the feedback of application action is not perceived. The user has attained closure and moves on to the next task.

Solution – the button widget must provide feedback when it invokes an application event, for example, a simulated key click. That is the actual event at the application must become a *perceived event* for the user.

Note, this is an expert slip, a novice would explicitly check the application feedback. This means that testing doesn't help to discover or rectify the problem.

4.5 General use of status/event analysis

Status–event diagrams may show different parts of an interface. For example, in an analysis of the arrival of email, the corresponding diagrams had timelines for the file system, the mail program 'mailtool', the screen, and the user. In all cases, status entities (the file system and the screen) mediated the events in the system. Not only was there a gap between the actual events on the screen and the user's perceived event, there was also a gap between the actual event of mail arriving (in the file system) and the perceived event for the mailtool. In general these diagrams are particularly powerful at tracing these gaps.

The application of the status/event distinction is wider than the use of status–event diagrams. Unfortunately, most dialogue notations (and other formal notations) deal primarily with one phenomena or other. However, we have seen that the interactions between status and event phenomena is particularly interesting, emphasising the need for both to be considered together. Status–event diagrams do this for specific scenarios, but one would like also have full dialogue notations dealing with status/event phenomena.

5 Summary

We saw that formal modelling techniques, although powerful and useful, require a high level of formal expertise. In order to 'give away' the benefits of this work to the typical human-factors practitioner less maths' intensive approaches are required.

Dialogue notations of various forms are often used during the interface design process. We have seen how simplified forms of the usability properties can be tested on dialogue descriptions, sometimes with automatic support. Furthermore, the dangerous states example showed how the dialogue description can form a focus for information from both the semantic level (what is dangerous) and the lexical level (what slips are easy to make).

Finally, we saw how status–event analysis can uncover expert slips which are very difficult to uncover during even extensive user testing. Status–event analysis is particularly useful where the interface is not purely reactive. Thus it is especially useful in open-systems and multi-user systems. The author is currently investigating the use of analytic techniques in the area of cooperative working.

Annotated bibliography

General

1. Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall International UK, Hemel Hempstead, 1993.

   The material in this paper is drawn largely from Chapters 8 and 9 of this book, which expand upon several of the areas.

Formal models of interaction

2. A.J. Dix. *Formal methods for interactive systems.* Academic Press, London, 1991.

   This covers the PIE model and many extensions and other models, including those on which status/event analysis is based.

3. M. D. Harrison and H. W. Thimbleby, editors. *Formal Methods in Human Computer Interaction.* Cambridge University Press, Cambridge, 1990.

   An edited collection covering a range of formal techniques.

4. Harold W. Thimbleby. *User Interface Design.* ACM Press, Addison-Wesley, New York, 1990.

   A wide ranging book which some extensive explicit formal material, and employing a formal approach to problems in much of its informal material.

5. A.J. Dix and C. Runciman. Abstract models of interactive systems. In P. Johnson and S. Cook, editor, *HCI'85: People and Computers I: Designing the Interface*, pages 13–22. Cambridge University Press, Cambridge, 1985.

   The original PIE paper.

6. B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, 1:157–202, 1982.

   A classic paper describing the formal specification of a display based text editor.

Undo

In case the reader's appetite for the fascinating area of undo has been wetted here are a few papers to read. In addition, see Chapter 2 and 4 of [2] and Chapter 12 of [4].

7. Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

   A formal analysis of undo in the context of group editing.

8. James Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages*, 6(1):1–19, January 1984.

   A classic paper analysing different forms of undo.

9. Jeffrey Scott Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, October 1984.

   Takes undo and redo to its extreme!

10. Yiya Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, May 1988.

    Informal analysis and review.

Dialogue

As well as the following, see Chapter 8 of [1] which describes dialogue properties in more detail and any book on UIMS.

11. H. Alexander. *Formally-based Tools and Techniques for Human-Computer Dialogues.* Ellis Horwood, Chichester, UK, 1987.

    Describes her SPI notation which is both quite powerful and very easy to read.

12. H. W. Thimbleby. Literate using for finite state machines. University of Stirling, 1993.

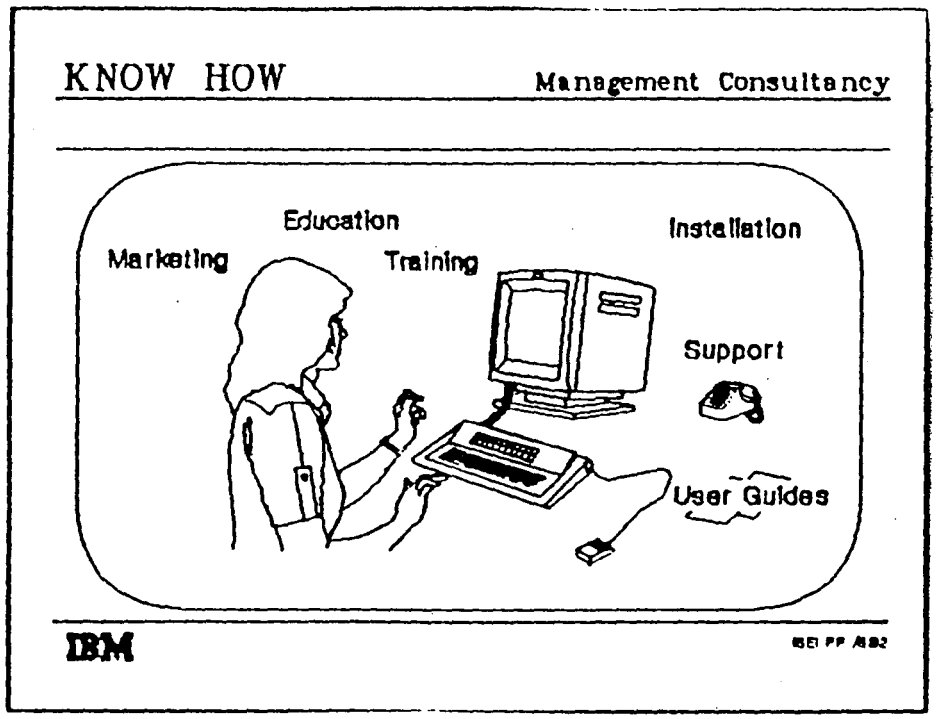
    Describes the Hyperdoc tool, which supports simulation, dialogue analysis and automatic documentation.

Status/event analysis

See Chapter 9 of [1] for status–event diagrams and Chapter 10 of [2] for its formal roots.

13. Alan Dix. Beyond the interface. In J. Larson and C. Unger, editors, *Engineering for Human-Computer Interaction*, IFIP Transactions A-18, pages 171–190. North-Holland, 1992. Proceedings of IFIP TC2/WG2.7 Working Conference, Ellivuori, Finland, 10-14 August 1992.

    Relates status/event phenomena to the timescales overwhich they operate and the concept of *pace.*

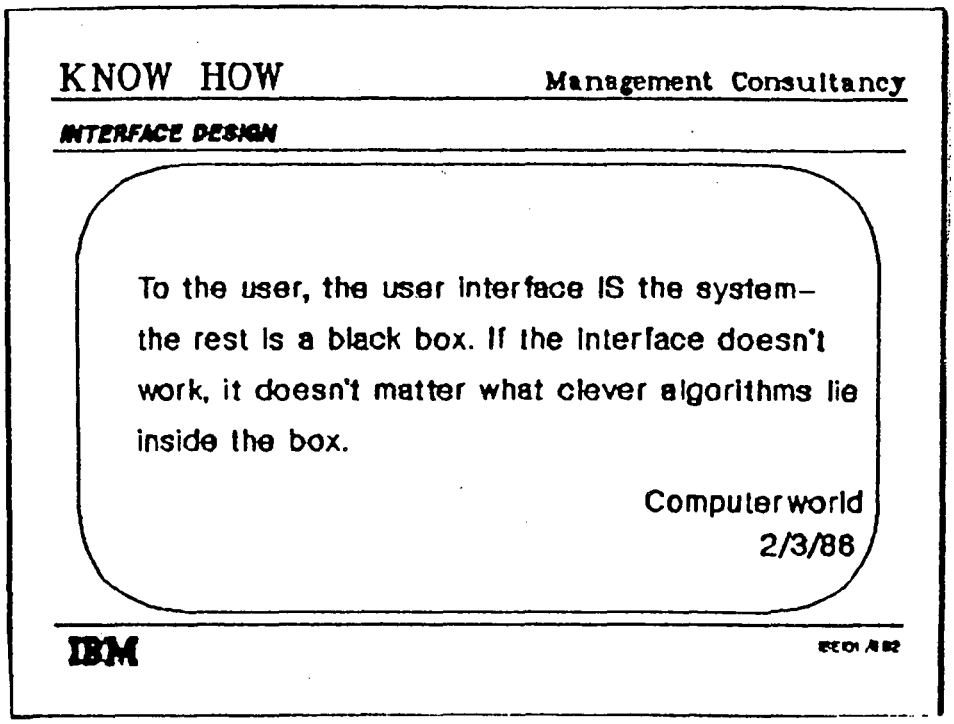

KNOW HOW
Management Consultancy
Education
Marketing
Training
Installation
Support
User Guides
IBM

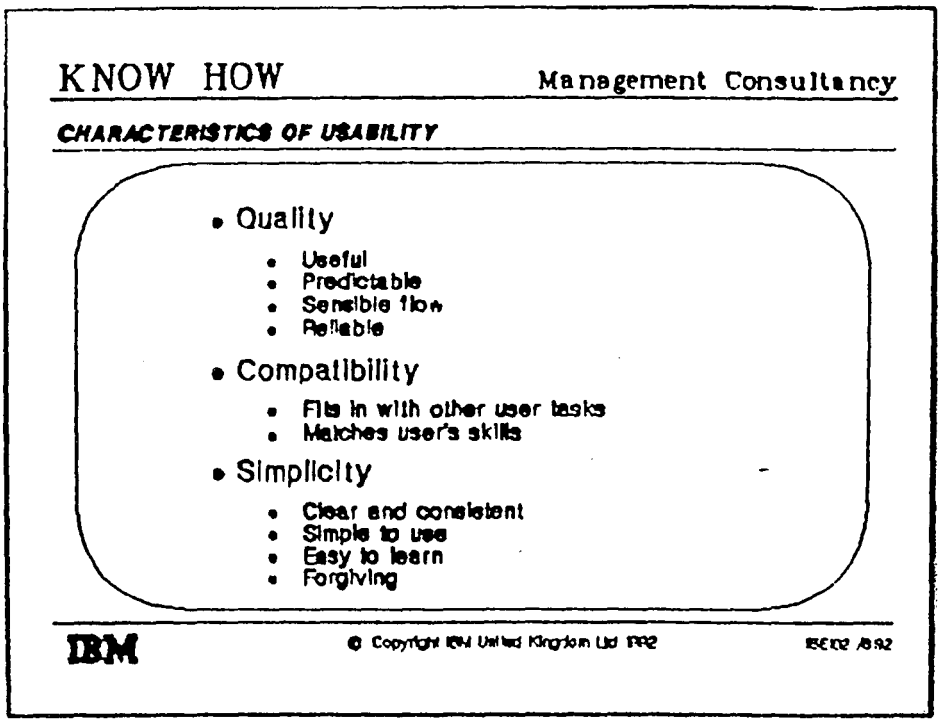
KNOW HOW — Management Consultancy

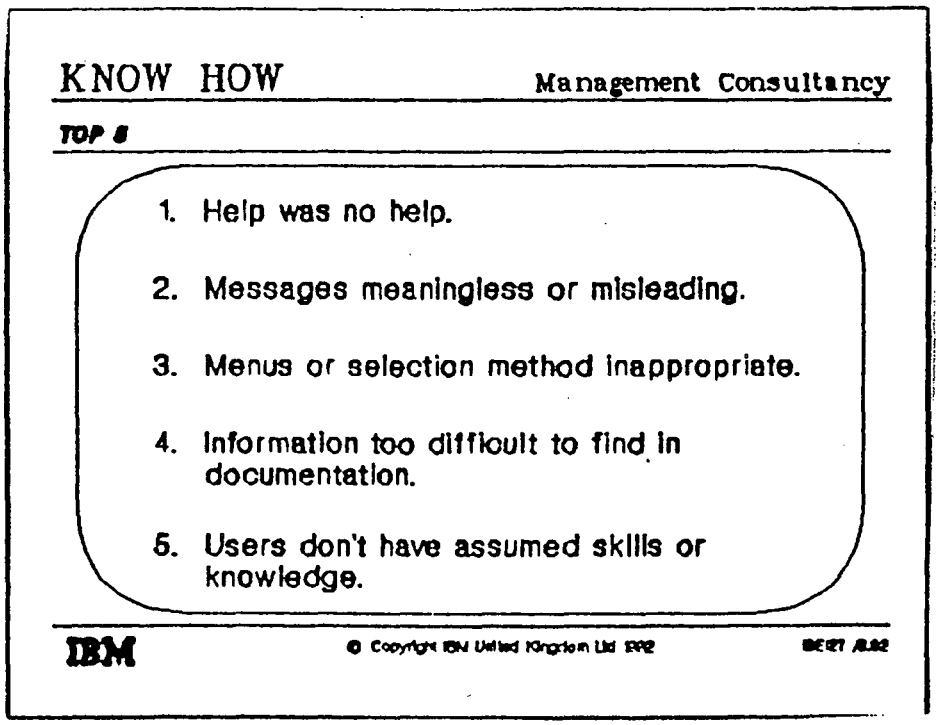
ORIGINS OF THE TOP 4

- Usability Health Check
- Usability Laboratory Evaluation
- Usability Field Assessment
- Usability Strategy Review

IBM

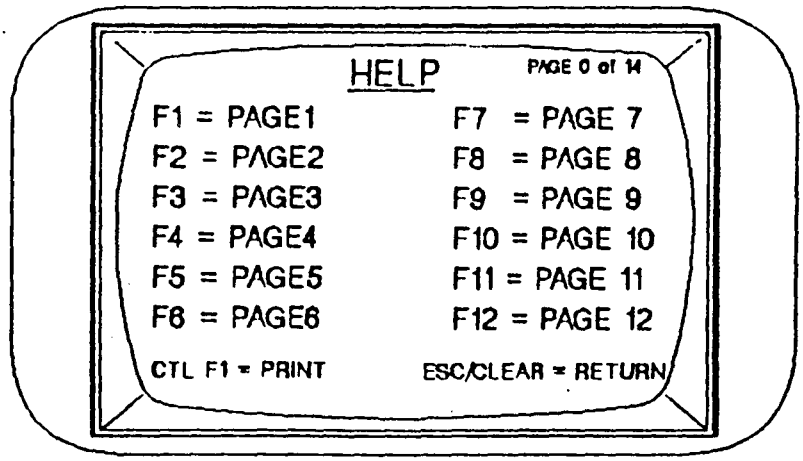

KNOW HOW
Management Consultancy
INTERFACE DESIGN
To the user, the user interface IS the system–
the rest is a black box. If the interface doesn't
work, it doesn't matter what clever algorithms lie
inside the box.
Computerworld
2/3/86
IBM


KNOW HOW
Management Consultancy
CHARACTERISTICS OF USABILITY
• Quality
• Useful
• Predictable
• Sensible flow
• Reliable
• Compatibility
• Fits in with other user tasks
• Matches user's skills
• Simplicity
• Clear and consistent
• Simple to use
• Easy to learn
• Forgiving
IBM
© Copyright IBM United Kingdom Ltd 1992

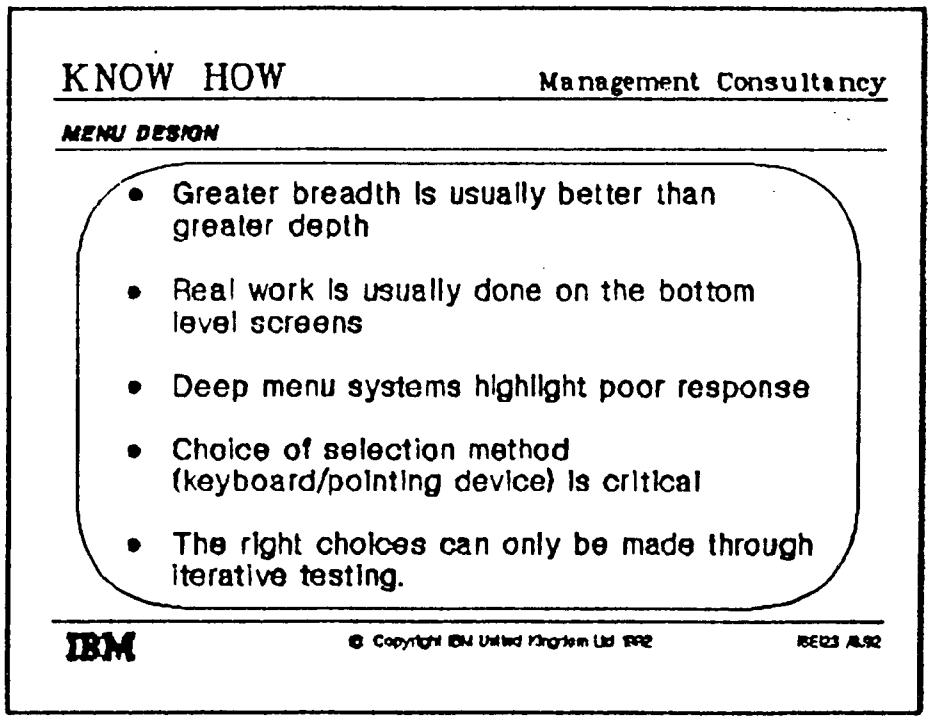

KNOW HOW
Management Consultancy
TOP 8
1. Help was no help.
2. Messages meaningless or misleading.
3. Menus or selection method inappropriate.
4. Information too difficult to find in documentation.
5. Users don't have assumed skills or knowledge.
IBM
© Copyright IBM United Kingdom Ltd 1992

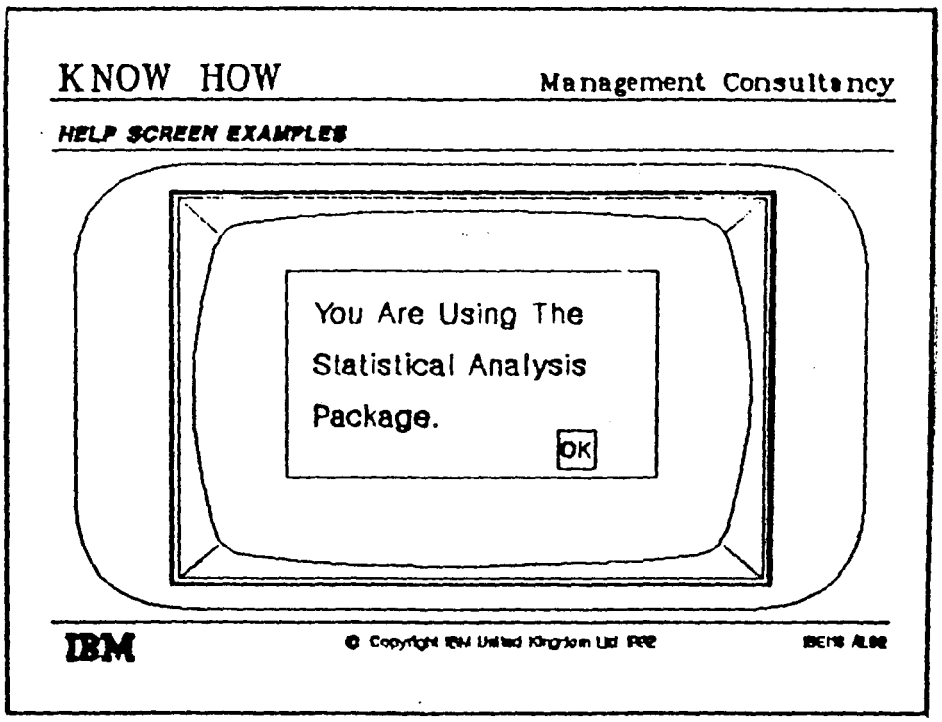
KNOW HOW — Management Consultancy

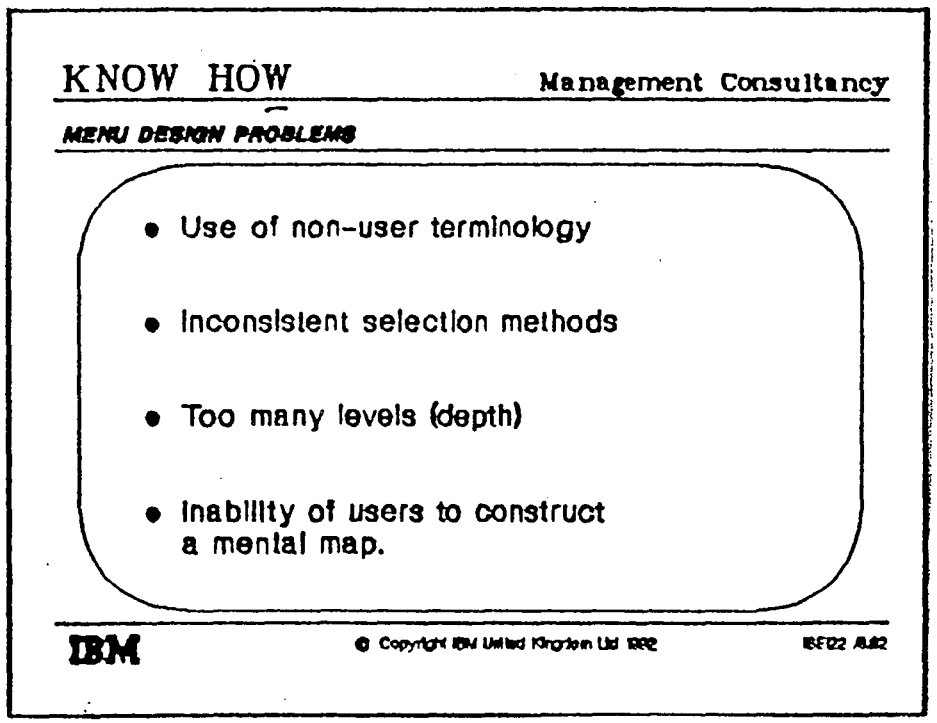
HELP SCREEN EXAMPLES

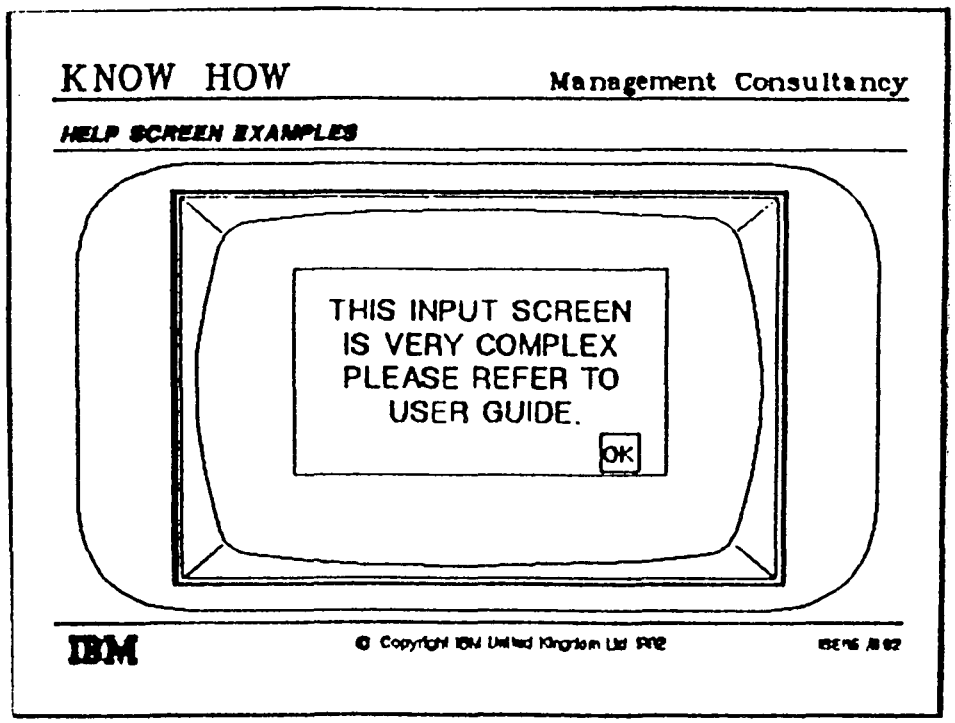


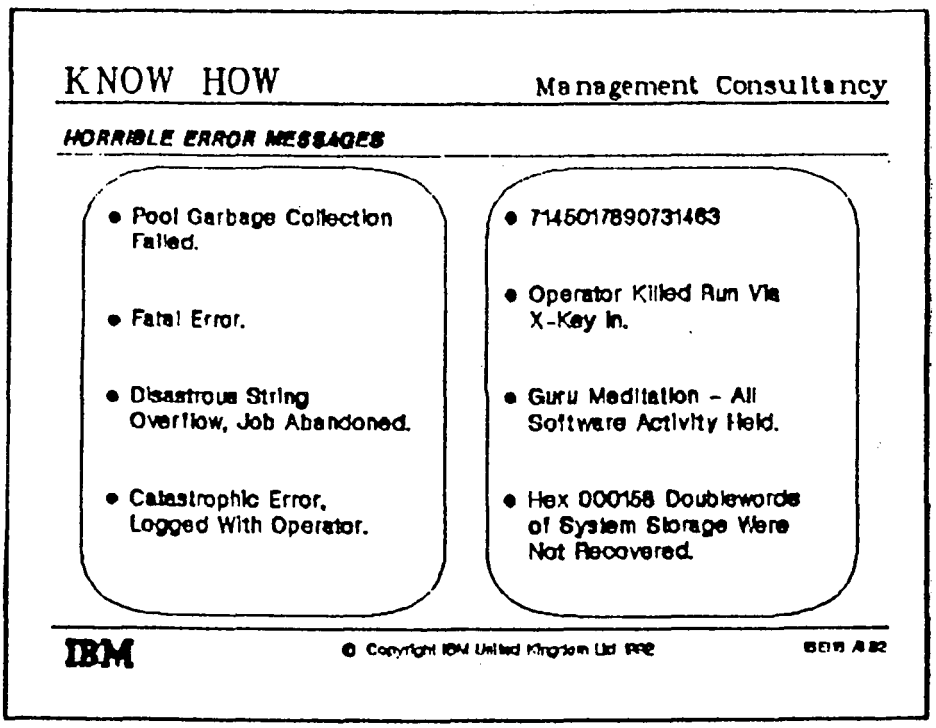
IBM IBEM3 AU92

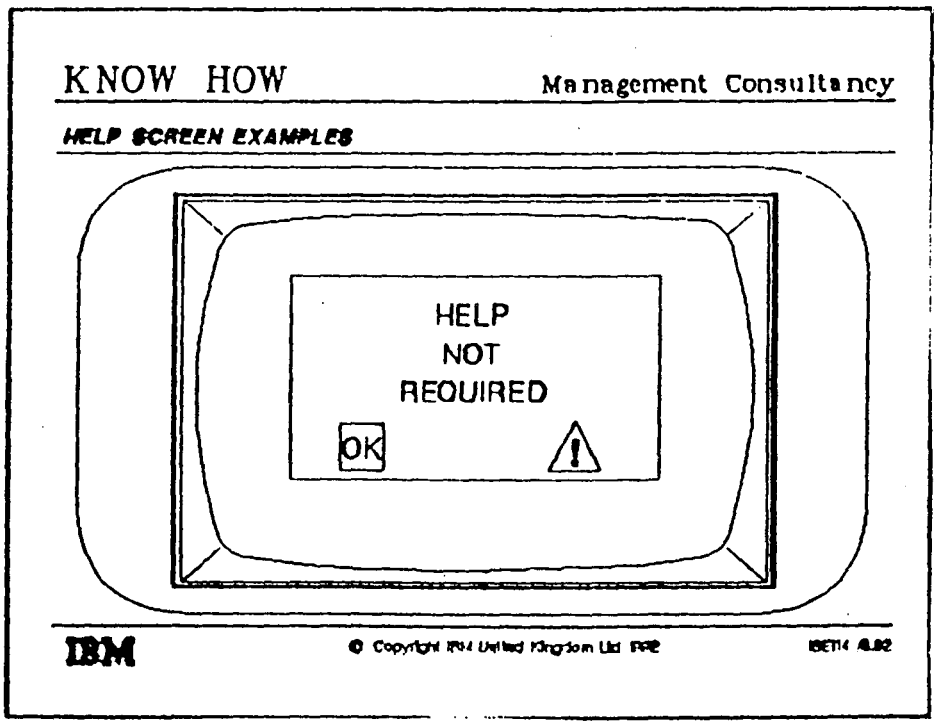

KNOW HOW
Management Consultancy
MENU DESIGN
• Greater breadth is usually better than greater depth
• Real work is usually done on the bottom level screens
• Deep menu systems highlight poor response
• Choice of selection method (keyboard/pointing device) is critical
• The right choices can only be made through iterative testing.
IBM
© Copyright IBM United Kingdom Ltd 1992


KNOW HOW
Management Consultancy
HELP SCREEN EXAMPLES
You Are Using The
Statistical Analysis
Package.
OK
IBM


KNOW HOW
Management Consultancy
MENU DESIGN PROBLEMS
• Use of non-user terminology
• Inconsistent selection methods
• Too many levels (depth)
• Inability of users to construct a mental map.
IBM
© Copyright IBM United Kingdom Ltd 1982


KNOW HOW
Management Consultancy
HELP SCREEN EXAMPLES
THIS INPUT SCREEN
IS VERY COMPLEX
PLEASE REFER TO
USER GUIDE.
OK
IBM
© Copyright IBM United Kingdom Ltd 1992


KNOW HOW
Management Consultancy
HORRIBLE ERROR MESSAGES
• Pool Garbage Collection Failed.
• Fatal Error.
• Disastrous String Overflow, Job Abandoned.
• Catastrophic Error, Logged With Operator.
• 7145017890731463
• Operator Killed Run Via X-Key In.
• Guru Meditation - All Software Activity Held.
• Hex 000158 Doublewords of System Storage Were Not Recovered.
IBM
© Copyright IBM United Kingdom Ltd 1982


KNOW HOW
Management Consultancy
HELP SCREEN EXAMPLES
HELP
NOT
REQUIRED
OK
IBM