THE
UNIVERSITY
OF CALGARY

Dialogue design notes

by

David R. Hill

Research Report 84/167/25

DEPARTMENT OF
COMPUTER SCIENCE

THE UNIVERSITY OF CALGARY
Department of Computer Science

Dialogue design notes

by

David R. Hill

Research Report 84/167/25

SUMMARY

This document presents an Informal discussion of some of the issues and steps In dialogue design. It Is Intended to be read in conjunction with Designing for human-computer Interaction: some rules and their derivation (D.R. Hill 1984) and A bibliography on human-computer Interaction (D.R. Hill 1984).

Man-Machine Systems Laboratory,
Department of Computer Science,
The University of Calgary,
2500 University Drive,
CALGARY. Alberta. Canada T2N 1N4

Some issues in design

Input handling and the implementation language

The use of fixed formats for input can cause real problems. Many of the languages used for programming interactive systems fall back to system level if the wrong kind of input (character Instead of numeric, for example), or the wrong amount of input (too many, or too few fields, or too many characters, for example) is provided. Ideally, a language and operating system designed for writing interactive systems should be used. BASYS, developed by Gaines and Facey (1977) is an example of such a language. It allows all system-level errors to be trapped and dealt with at the dialog level. Coupled with an appropriate start-up file (such as the .login facility under UNIX, or the start_up.ec of MULTICS), and a method of logging out directly from the application, this can allow the application user to be protected from everything that is not strictly connected with the dialog itself. Failing that, however, great care should be taken to avoid system errors. At a brute-force level, every input can be read as a (virtually) unlimited character string, and the processing into fields, numbers, etc. can be handled by the dialogue program which can, therefore, detect, correct and interpret all manner of strange inputs, correct inputs, errors and omissions, etc. Of course, some languages have poor string handling facilities too. The combination of poor string handling and absence of system error traps means that you should avoid the language as a vehicle for interactive system implementation, and hand in your notice if you are compelled to go ahead anyway. Most supervisory staff can be convinced of the need to acquire proper facilities if they can be shown the difference. You could even write your own implementation language, if proprietary interests are at stake. The last named solution requires considerable understanding of what is required and obviously acquiring or writing suitable facilities means extra time and cost the first time through.

Information integrity; unified databases and 'viewing algorithms'

In many interactive programs (perhaps most), quite apart from the dialogue problem, there is a database problem to be solved too. Information has to be entered and updated, new information produced and displayed, and so on, even if only in connection with help facilities.

As soon as the same or related information exists in a number of forms, dispersed amongst several different structures, it is all too easy for the different embodiments to get out of step, especially in the event of a system crash whilst updating. The program design and debugging needed to avoid the problems can become tedious and expensive. One good approach is to keep just one carefully structured database, using linked list techniques, and to generate any displays, or retrieve any information from that one list, as needed. In a multitasking environment, mutual exclusion from the one database, coupled with provisional assignment of the resources represented in it when a tentative request is initiated, with confirmation of allocation making the provisional allocation permanent, can ease problems.

Thus, when it is requested during checkin, the seat plan for the airline dialog can be displayed by a routine which traces through the data structure and picks out the seat allocation data, filling out the plan with current data as it displays it. The status of unallocated seats on the display would need to be frozen in the database (mutual exclusion), until the current passenger had been processed. Seat allocation Is normally done at check-in so there should not be too much of a hold up for other passengers at other check-In terminals. If the delays did prove a problem, the seating could be broken Into blocks associated with different check-in positions. Alternatively, the display could be continually updated, but that would produce Interesting 'race' effects. The user might find it possible to select an apparently unallocated seat but not get it, in a race with a selection from another terminal position. This is clearly not a problem in a pilot implementation, using only one terminal.

A routine that traces through a database to produce a graphical display is called a viewing algorithm. More than one viewing algorithm may operate on the same database, yielding different views. The notion of a trace algorithm is not limited to the production of graphical displays. Trace routines could be devised to search through a database to extract and output (in suitable form) any aspect of the data. Conventionally, if the primary purpose of the database is graphical display, it is called a graphical database. In the case of a graphical database representing an electronic circuit—as for the MINNIE system (Spence 1975, 1977; Spence and Apperly 1982), another trace algorithm could pull out all the component values and connections, and simulate the circuit behaviour, 'displaying' the results as a graph of circuit response.

But even a database that is not primarily graphical can be thought of in terms of the various views that may be obtained of the data by different trace algorithms. For the airline seating database, for example, there could be a seat-list trace, a passenger record trace, the stand-by list trace, and so on. At any given moment, however, there would be one source for all old information, and one destination for all new information. This makes control and consistency much easier. Needless to say, there is a price to pay—the complexity of the unified database.

VDTs and screen management

The limited screen size of standard VDTs can be a problem, as can the inadequacies of screen management packages—especially the limitations they may place on the form of information entry, the management of errors and error messages, and so on. Taking care of all possible problems is difficult, one of the important difficulties of dialogue design in fact. A screen management package is supposed to be a generalised tool for such design, so that it faces the same problems. Unfortunately many packages are written without a complete understanding of the requirements of dialogue design and/or without a full commitment to achieve the necessary degree of comprehensiveness and perfection. In these circumstances the window system may be imperfect, or may interact with the operating system in ways that virtually preclude meeting all the goals of interactive system design, if it is used. Thus an early window system on MULTICS was subject to corruption when too much information was entered, causing an automatic scroll. The same system did not handle input on a character by character basis, and generated line feeds inconsistently.

The variation in terminal types and access causes problems. The best systems must be able to configure themselves to extract the best performance from whatever terminal facilities are implemented for the user's terminal. If this is not practical, the designer winds up designing for the dumbest terminal that will be used, and it becomes difficult to handle screen formatting both quickly and effectively. Pagination into screen window size will be needed for output, with operator control over paging. A programmable intelligent terminal with a bit-mapped screen (the Corvus Concept for example) can overcome some limitations of restricted host systems, at the expense of yet more programming, by doing some of the dialogue handling locally. This leads into distributed computing, software migration, and many of the problems that the Jade research project is tackling (Jade is a major research project within the department of computer science at the U of Calgary).

Error handling and input alternate forms, defaults, reasonableness and recovery

Introduction: Error messages and error handling are of extreme importance in interactive system design. Humans make errors, even when highly skilled, highly motivated, and conscientious. Deal with errors in a positive and helpful manner. If possible, identify the cause of the problem as narrowly and succinctly as possible, and give explicit information on how the error may be avoided or corrected. Use neutral terms in all parts of the dialogue, meticulously avoiding emotionally loaded terms. But in error handling it is especially important to be clear, non-threatening, and supportive, without being unctuous, patronising or rude. The objective is to communicate, not insult. There is an excellent little paperback, written to help people communicate their needs effectively and without counter-productive emotional upheaval, even in the face of lack of communication skills on the other side (Smith 1975). The technique of assertive communication is described, and there are many lessons in there for the interactive systems designer as well as for the general conduct of human communications.

In checking answers, it is dangerous to assume that if the answer is not one of the first n-1 alternatives, it must be the nth. of n alternatives. Part of checking input is to ensure that it is syntactically correct and 'reasonable'. This is tied in with defaults, reasonableness checking, and allowance for alternate forms, as well as error detection and handling.

Alternate forms and recovery: A good example of the need for alternate forms occurred on the early version of the PERQ (a workstation designed explicitly for research on personal scientific workstations at Carnegie-Mellon University -- the SPICE project). I was being given a demonstration, and the demonstrator was faced with the message: "Please enter the date" as part of the login procedure. He was unable

to come up with anything that the computer would accept, and the demonstration was long delayed as a result because “Help” was inactive during login. There is absolutely no reason why a basic requirement of interface specification should not include a requirement for all reasonable alternatives for the date to be legal. Since dialogues should provide feedback, and this can be in (selectable) standard form, the user can re-enter the date if it is misinterpreted. In dialogue it is desirable to help users by allowing flexibility in forms of expression. Reasonable alternatives should be allowed wherever possible. Not only dates and the like, but any kind of input can allow synonyms, abbreviations, upper and lower case letters and close mis-spellings. Ambiguity turns out to be surprisingly rare and can be resolved by a brief addition to the dialogue. Thus full command words, or any reasonable abbreviation, should be allowed, with flexibility in specifying options. Some constraints can be included to provide redundancy and reduce the likelihood of ambiguity or allow error detection and perhaps correction. It is nevertheless worth providing a facility to help the operator who makes a trivial error in a complex command. The simplest is to allow the command fine itself to be edited and retried, if the first invocation does not work. Alternatively, the rather sophisticated facility introduced in the Bolt, Beranek and Newman TENEX operating system may be implemented, namely the DWIM facility, which stands for ‘Do What I Mean’. DWIM was used when a command was not recognised. It caused the system to determine the closest approximation to the command typed, that was a valid command, and execute it. There was also an UNDO command so that when the user discovered the true effect (by feedback, of course) the effect could be completely undone, if it was not what the user intended. An UNDO command is a very powerful tool that makes all actions reversible and considerably reduces anxiety, thus further reducing errors. However, a general UNDO facility, especially one that can be applied repeatedly, takes a lot of effort to implement and can slow response as well as consuming a great deal of memory space. If an UNDO facility is impractical, care should be taken to double check commands having serious and irreversible consequences. Apart from avoiding dangerous consequences, as in a real-time control dialogue, a dialog should avoid loss of previous effort, or generation of unnecessary work (usually keystrokes). One general philosophy behind much of this type of thinking is to minimise both keystrokes and memory load whilst making the system very forgiving. This helps to avoid many human failure modes.

Keystrokes and buffering: In the interests of minimising keystrokes, it might be desirable to avoid the use of the carriage return (or enter key). This is only possible if the system is continually monitoring the input, on a character by character basis. Although this may involve considerable system overhead, it is very effective in creating helpful dialog effects—such as immediate error detection, automatic filling out of command word abbreviations, fast response, and keyboard acceleration by prediction. One problem that may then be created is that of having ‘unused’ characters floating around—characters typed after the system has decided what the input was supposed to be or after it has detected an error. To combat this it will be necessary to flush out the input buffer as the last action before waiting for new input, and after printing the next prompt. This is not totally foolproof, but usually works for most users, who tend to stop typing once a reasonable number of output characters they didn’t type have been generated. In any case, for input, a buffer is essential. There are few things so disruptive to typing rhythm as finding the odd character does not get processed because the user happened to beat the system. Even slow typists occasionally type two successive characters in very rapid succession.

For experts who are good typists, the input buffer may need to be quite large, especially if there will be system delays for output or processing. One interesting feature that can be added in this case allows, forestalling of prompts—system prompts are not typed out if the responses that would be required are already in the buffer. In the case of line-at-a-time input, some of the benefit of this can be obtained by allowing commands to be concatenated on a line, using separators. For keyed input to a system using speech prompts, forestalling of prompts is especially valuable because of the slow output speed for speech.

Defaults: A further measure to reduce keystrokes is to provide defaults for responses required from a user. If the response possibilities are restricted, and there is in any sense a ‘usual’ response, allowing the user to select the default by making a null entry (by just hitting the return key, for example), can save a great deal of effort. If an entry other than the default is required, the user may type in a different response, and

the default is ignored. The user loses nothing in the latter case and gains considerably, by frequent savings, in the former case.

Reasonableness checking: Reasonableness checking is a feature that checks a user response against a range, stored somewhere in the database, that suggests reasonable limits on values for the response. It is a form of error protection, and can prove annoying if overdone. It can avoid embarrassing errors in requests (such as ordering materials) and, like many features designed to help, perhaps should be selectable as part of the user's chosen working environment, along with verbosity, confirmation requests, and the like.

Uniformity and consistency: learning, help and excursions

The designer should aim for consistency and uniformity in dialogue design. As Gaines & Shaw (1983) have succinctly stated: all terminolgy and operational procedures should be uniformly available and consistently applied throughout all system activities. It is possible that certain operations, facilities, or terminology may simply be inappropriate at certain places in a sub-dialogue, but, with this proviso in mind, it should be possible to access all facilities all the time (which is uniformity) in the same way (which is consistency). Darlington, Dzida and Herda (1983) have introduced the notion of an excursion tour, defined as an information gathering sequence of operations that enables a user to learn the commands that implement the dialogue proper. It is related to the distinction in cognitive theory between knowing and doing. Dialogue excursions allow the user to extend his or her knowledge without leaving the system, so that 'doing' may be continued with minimum disruption once 'knowing' has been updated. The simplest form of excursion tour is represented by basic help facilities, which certainly should be available all the time. However, the model may be extended to a much more sophisticated form of self-help where, for example, the user may peruse files in some arbitrary directory whilst in the middle of an editing session. In this way interactive deadlock can be avoided. An interactive deadlock is a situation in which a user cannot proceed because a command to reach an essential subgoal within the dialogue either does not exist, or cannot be discovered within the system. Wide-ranging excursion facilities make the system facilities available all the time in a very general way, and may allow alternative paths or appropriate commands to be found. If the user is an expert programmer, the excursion facility can allow access to the whole power of the system, as does the shell-escape on UNIX. Incidentally, it is not inconsistent to allow query-in-depth as defined by Gaines *et al.* (Gaines & Facey 1975; Gaines & Shaw 1983). On the contrary, it is highly desirable. In a system providing query-in-depth the user may type a '?' anywhere a response is expected and the system will respond with brief information about the response options. As additional '?'s are entered, the helpful information is elaborated. Ultimately, a user may be referred to an on-line manual, available by means of an excursion, with the user placed at what seems the most helpful place to start with. At each stage, the help provided is chosen to be as closely relevant to the actual context as possible, and in the briefest form. In giving information to users, it is highly desirable to give a little information, and only add to it as required. It is very frustrating to get a complete description of how the system is operated, or to have to negotiate a deep hierarchy of menus, every time a spelling mistake is made.

Modelling and metacomments

As well as helping the user to form an accurate model of the system, related to his or her previous experience, there is value in the system modeling the user. Some parameters (error rate, response time, the user's deliberate choice, ...) can be used by the program to infer how to treat the user (naive, inexperienced, secondary, ..., expert, ...), and prompts, help, etc. adjusted accordingly. The user should always be able to force the system, however. Many initial attempts at dialogue treated the user as a complete idiot, who could only understand basic English (or whatever) and simple child-like constructions. It is, as usual, partly a matter of knowing the user, and his or her expectations. But the early problems also arose, in part, from a failure to understand the difference in behaviour needed from a system, once it was interactive. Like a conversation, an interactive dialogue can economise, on the basis that failure to understand can be signalled by either participant. This comes under the heading of metacomments in the dialogue—a phenomenon that is important in human-human dialogue. Metacomments are important in keeping the communicating entities 'in step'. The ideas and consequences are very ably explored in a paper by Thomas (1978). Metacomments allow a participant to say things like: "Would you mind rephras-

ing that", or: "Your are going too fast for me", or: 'Do you really mean ...'. Providing the ability to handle a wide range of metacomments could prove a daunting task, but even present systems clearly provide some facilities, as when a system asks for confirmation of a serious action. A good metacomment facility could give a real feeling of control and communication, though it seems equally clear that a poor one could prove tiresome and frustrating. The secret is not to be too ambitious, but to consider carefully what can usefully be provided.

A rather simple form of metacomment allows the user to interrupt a long dump of information, perhaps invoked accidentally or mistakenly ("Well, I really didn't want all that stuff"), or to leave a session and pick up where he or she left off ("I'm tired, do you mind if we continue this tomorrow").

The form as a prompt

Forms provide an excellent aide memoire, reminding the user what is required in an assertive and effective way. A form is an elaborate structured prompt that allows the user to tackle items in the most convenient order. Putting a forms-based dialogue on a VDU avoids some of the problems of question-answer style dialogues, and gains the same advantages as a form if an addressable cursor and protected fields are available. I have seen a forms-based dialogue used to construct process-control programs and associated mimic diagrams in a real-world chemical plant, for example.

In considering the terminal, thought must be given both to ease of use, and to cost For an airline reservation and check-in terminal, special hardware could be considered, since many thousands would potentially be used, and development costs could be written off over a reasonable number, bringing the total cost per terminal down to something approaching standard terminals. Then the commands used could each be assigned special keys, acting as a built-in menu and reducing keystrokes. In other cases, there is the possibility of using a standard keyboard, with an extra function keyboard, or even a standard terminal having function keys—especially if the functions keys are fully programmable. Graphics are common now, and offer the possibility of soft labels for function keys, so that a small number can cover many possibilities, if only a few are appropriate at any given time.

Steps in the design process

Who is it for; what do they do?

For whom are you designing the system. Find out, and arrange to talk to a variety of potential users about their needs, and how they think about the problems they have to solve. Get hold of documents and manuals that describe what they have used to help them in their work up till now. Make a list of the basic terms and facilities they have used. If possible, watch them doing their job, see what difficulties arise, and see if they really do what they say they do. Try the system yourself (whether manual or on a computer). What is good about the existing system and what is bad. Make brief but informative notes. It is only rarely that you are likely to be involved in a totally new system, but if you are, you should still talk to potential users, representing a reasonable cross-section, and find out how they think and what their relevant experience is. Some notes will still be possible. You will simply have more work later.

Consider what other specific subtask facilities should be provided to help the user in meeting his or her task goals. Make two lists: the facilities needed, and the concepts involved. The former list will imply certain methods. Look for structural possibilities in both lists, and restructure them, grouping related subtasks or subconcepts. Think about mnemonic possibilities (text, icons) for commands and concepts. It may help to write out the material in the form of questions and notes as a starting point

You have now made an outline of the System capabilities section of the user manual, and should be able to define, in detail the purpose of the system. Now, with the help of detailed design principles, outline the overall flow of the dialogue and its sub-dialogues. The procedure is quite similar to top-down design of a program. Augmented transition diagrams (there is an example in Foley & Wallace (1974)) are a recommended method of showing the overall structure of the dialogue relative to the 'home state'. Keep it modular, or it will get just as much out of control as a great chunk of unstructured program. Define the pos-

sibilities under which various branches will be taken, and use subgraphs as needed to keep things simple. A subgraph is akin to a subroutine and may cover, for example, a general error handling subdialogue, or an excursion. Jacob (1983) has commented on the value of state transition diagrams as opposed to BNF representations, and gives a discussion and some simple examples.

Dialogue style and screen management

You now have a major decision to make, concerning the style of dialogue. This should be appppropriate both for the user and for what the user is trying to do. If graphical display is required, you will need to consider the specification of the terminal. Some (e.g. the Visual 550) allow fairly extensive graphics, without the expense of a full scale graphical workstation. If the interaction itself will be in the graphical domain, you will need to worry about graphical input devices suited to the kind of actions involved (Foley & Wallace 1974) and are likely to be involved in the design of a sophisticated graphical dialog based on a full graphics workstation. If a VDU is adequate, you still need to consider the keyboard, which should be detachable, familiar, and equipped with function keys. You may consider a touch overlay on the screen if selection of displayed items is going to be part of your dialogue. A good example of such a screen exists on the Victor 9000. You need to consider how much intelligence there is in the terminal and how much is needed. You may find certain dialogue techniques are impossible unless you have a full scale microcomputer and bit-mapped screen to double for a terminal. For a VDU, you must decide on screen handling. It is usually true that a dialogue using formatted screens, whether forms, menu or prompt based, is the best solution. Allowing material to drift by and occupy arbitrary locations as it scrolls sequentially out of the computer is sirnply not good enough. If, for example, a scrolled question-answer type dialogue is appropriate, it is still worth confining it to a restricted area of the screen, with other areas designated for system state feedback, error messages, and the like. This is most easily done with screen management tools.

Design screen and/or text formats for major sections of the dialogue and try out some bits and pieces of dialogue, modifying them till they fit your understanding of how the user wishes to be presented with information and proceed with task execution. The existence of dialogue tools and screen management tools can be very helpful here, but few good systems currently exist On the basis of these experiments, and with the benefit of feedback from the user, who can be shown some samples, draft the transaction details part of the user manual, and fill in the missing parts that you have not yet tried. If you have' access to the kind of dialogue prototyping tools mentioned, you may be able to incorporate hard copy of test screens as part of your draft. However, decent sketches are almost as good, even if they may take more trouble to produce. Try writing down some complete sample dialogues at this stage to see if the dialogue allows the various possibilities to be covered conveniently.

Completion of the user manual

At this stage, you should be able to complete the draft of the user manual, adding the 'Review of system' section after several critical readings of the manual by yourself, and by some of the users. If you find you cannot write the user manual, it could be that you don't know what the user's are trying to do. That could mean you need to do quite a bit of research first

In implementing any system you will find that an amazing amount of code needs to be devoted to input checking, error handling, and dialogue script. Even apart from program and data volume, there is the problem of phrasing things correctly, choosing sensible defaults, bullet-proofing the system, showing what is going on, choosing mnemonics, and so on. If the design phase has been executed conscientiously, much of that work will have been done. Experience confirms just how helpful proper design discipline can prove in facilitating the implementation.

References

Darlington, Dzida and Herda (1983) The role of excursions in interactive systems. *Int. J. Man-Machine Studies* **18**, February, 101-112.

Foley, J. & Wallace, V. (1974) The art of natural graphic man-machine conversation. *Proc. IEEE* **62** (4), April, 462-471.

Gaines, B.R. & Facey, P.V. (1975) Some experience in interactive system development and application. *Proc. IEEE* **63** (6) June, 894-911.

Gaines, B.R & Shaw, M.L.G. (1983) 'Dialog engineering', in *Designing for Human-Computer Communication* (eds. M. Sime and M. J. Coombs), Academic Press, London) pp. 23-53.

Gaines and Facey (1977) 'BASYS -- A language for programming interaction.' in *Proceedings of Conference on "Computer Systems and Technology", IERE Conference Proceedings* **36**, March, Univ. Sussex, UK, 251-262.

D.R. Hill (1984) Designing for human-computer Interaction: some rules and their derivation. *Univ. Calgary Dept. of Computer Science Research Report 84/166/24*, 15pp.

D.R. Hill (1984) A bibliography on human-computer interaction. *Univ. Calgary Dept. of Computer Science Research Report 84/168/26*, 13pp.

Jacob, R.J.K. (1983) Using formal specifications in the design of a human-computer interface. *Communications of the ACM* **26** (4), April, 259-264.

Smith, M.J. (1975) *When I Say No, I Feel Guilty: How to Cope - Using the Skills of Systematic Assertive Therapy* (Bantam Dell/Random House: New York, ISBN: 0553263900, ISBN13: 9780553263909), 324pp.

Spence, R. (1975) Man, computers and creativity: the dialogue problem. *Armstrong Memorial Lecture,* Columbia University, New York, 25th April (Department of Electrical Engineering, Imperial College, London, UK).

Spence (1977) The Interactive-Graphic Man-Computer Dialogue in Computer-Aided Circuit Design. *IEEE Transactions on Circuits and Systems*, Vol. **CAS-24**, (2), February.

Spence, R. & Apperley, M. (1982) Hierarchical data structure in interactive computer systems. *Institute of Electrical Engineers Conference on Man-Machine Systems*, UMIST, Manchester, 6-9 July. London: Institute of Electrical Enginwers Publication number 212.

Thomas (1978) A design-interpretation analysis of natural English with applications to man-computer interaction, *Int. J. Man-Machine Studies* **10** (5), November, 651-666.