KNOW YOUR TECHNOLOGY
or
CAN COMPUTERS UNDERSTAND DESIGNERS? [1]

Aart Bijl
Edinburgh Computer Aided Architectural Design
Department of Architecture
University of Edinburgh
20 Chambers Street
Edinburgh EH1 IJZ

August 1983

Any great expansion of the population of computer users, embracing architects and other ordinary people, will happen only if we change from current computing technology to radically new software technology.

Criteria for new technology are discussed, with reference to Inadequacies of current technology; we should strive for computers that can understand people. Logic programming is described as one development towards this goal, illustrated by the example of Prolog serving as interpreter of user demands and supporting partial and charging logical models of user activity.

Architects can choose computing options now that will put then on a path leading to future new technology. Choice is explained, favouring a software environment that is used by researchers and also supports immediate and practical computer applications.

Lessons are drawn for architectural education, to prepare for change that will take place during a student's 40-year working life.

THE WORLD OF COMPUTER USERS

When a designer is invited to use a computer, some part of the designer's normal activity will need to have been encapsulated in the computer. The visible manifestation of that encapsulation must then be capable of being understood and assimilated by the designer into the rest of his or her activity outside the computer. Here we have the essential goal of computer aided architectural design, and we also have the clues to problems which still stand in the way of successful CAAD systems.

1 This paper is a modified version of the similar paper "Can Computers Understand Designers?" presented at PARC83, London, October 1983.

We know that designers have to be responsive to the volatile world of all people who make demands on buildings, and that designers in turn present a volatile world of procedures to computer systems (Bijl 1982). The argument that design procedures are necessarily idiosyncratic to individual designers, design offices and design projects is illustrated in figure 1. The idiosyncratic nature of design procedures is a consequence of the world of all people to whom designers are answerable, to the world of butchers, bakers and candlestick makers.

On the other hand, we have a body of computer programs that represent a considerable investment in research and development and which is intended to be used by designers. What prevents designers using these programs? Generally, these programs have grown out of the tradition of computer applications in disciplined scientific and technological fields, in academia and industry, and in commercial fields such as accounting and banking. This tradition aims computer applications at fields in which demands on computers can be explicitly identified and where demands stay constant for long enough to allow system analysts and program implementers to do their work. There can be a reasonable expectation that finished programs will perform tasks which users will recognise as being valid.

In the volatile world of designers, this expectation is not reasonable. Users will argue with the view they are given of computer resources through the programs that are offered to them. The programs will not do what they want. The arguments became louder the more nearly programs are successful; programs that are good enough to be used will attract the most criticism and thus we get the third law of computer programming (Dickson 1978); if a program is useful, it will have to be charged.

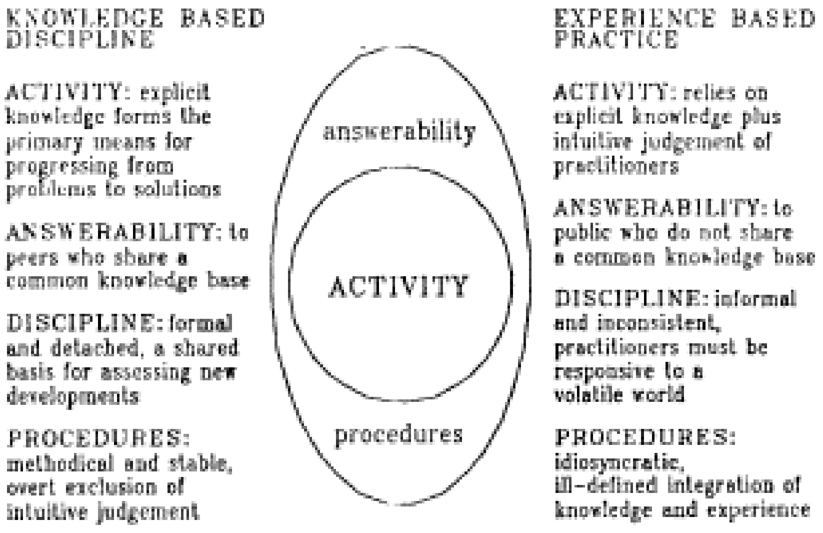
NEED FOR NEW TECHNOLOGY

Broadly, there have been two responses to this conflict between designers and computers. The first is to say that existing computer programs represent substantial achievements and that users have to he changed, educated, so that their work-practices will present the necessary and stable context for existing programs. Thus we get the attempts to coordinate, harmonise and standardise design practices nationally and internationally across Europe (CIAD et al 1979).

The second response is to say that the volatile world presented by designers has to be accepted as given, and that any conflicts between designers and computers point to inadequacies in computing technology. This argument recognises that the technology is still at an early stage of development and is going to change. The demands made by designers are then accepted as a welcome influence of the further development of new technology.

This latter response fits into the new national and international programmes for Advanced Information Technology (AIT) and Intelligent Knowledge Systems (IKBS) following the Alwey Report (1982), and the Japanese Fifth Generation Computers project (JIPDEC 1981)). These efforts, and particularly the Japanese project, take a long term view that foresees a vast expansion of the population of computer users, beyond the present relatively small group of specialist computer users and including ordinary people; we may choose to see the latter as being represented by non-specialist designers. The Japanese see this expansion as essential, to provide the economic base for production of new hardware that we already know how to make.

This link between commercial interests and the variable needs of ordinary people will ensure that future computers will look very different to Pets and Apples, or even the most upmarket minis. We can have a fair idea of what future computing systems will look like, but architects who wish to get into computers now will, quite reasonably, be wary of jam tomorrow. The promise of new technology can be important in helping architects to make practical choices now that will put them on a path leading to future better systems.



KNOWLEDGE BASED DISCIPLINE

ACTIVITY: explicit knowledge forms the primary means for progressing from problems to solutions

ANSWERABILITY: to peers who share a common knowledge base

DISCIPLINE: formal and detached, a shared basis for assessing new developments

PROCEDURES: methodical and stable, overt exclusion of intuitive judgement

EXPERIENCE BASED PRACTICE

ACTIVITY: relies on explicit knowledge plus intuitive judgement of practitioners

ANSWERABILITY: to public who do not share a common knowledge base

DISCIPLINE: informal and inconsistent, practitioners must be responsive to a volatile world

PROCEDURES: idiosyncratic, ill-defined integration of knowledge and experience

Figure 1: Design, discipline or practice?

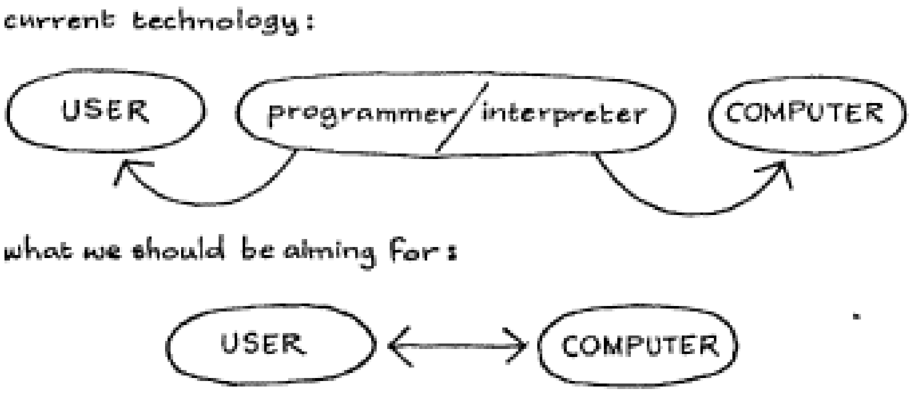


Figure 2: Users' access to computer resources

CRITERIA FOR NEW TECHNOLOGY

What are the criteria by which we will recognise the emergence of new computing technology that can serve the varying demands of ordinary people? How can we judge whether some new development is significant?

The single most important criterion is the ease with which something can be undone and reconstructed, to allow you to change your mind. Design, like many ordinary human activities, has to be responsive to unforeseen situations presented by any other people. It is not practical to devise a single and complete design system that will operate consistently in different design contexts. Nor can we be confident about fragmenting design into sub-systems that will be recognised as equally valid among different designers (Bijl et al. 1979). Whatever system is implemented in a computer for use by designers, and however enthusiastic the system authors and system sponsors are, if the system is used on live design projects we can be sure that the system will need to undergo charge. So we need a computing environment in which it is easy to make changes to systems that are offered to people.

What has to be capable of being changed? What are the elements of a computing system that we need to be conscious of, which inhibit change?

To answer this question, we need to establish a view of computing systems. These systems may be described in terms of three main elements.

1. Logical model; this represents how you think about the task that the computer is going to help you do, an abstract representation of the tasks of designing and analysing things.

2. Software system; a means of implementing a logical model in terms of descriptions and instructions that can be "understood" by computers.

3. Hardware system; the electronic and mechanical devices that constitute the physical objects we recognise as computers, whose operations are invoked by software.

The logical model, encapsulated in software which in turn governs the operations of the hardware, has to match your expectations of the tasks you want to see a computer perform. It is the abstract logical model that needs to be capable of being easily changed.

Developments in software and hardware technology should be judged by how they improve the ease with which logical models may be charged. The need for improvement should be apparent if we consider how people currently gain access to computer resources.

Figure 2 shows potential computer users, architects, on the one hand, and computer resources that are potentially able to assist users on the other, with programmers serving as intermediaries. The programmer offers his knowledge of the internal workings of computers, and interprets what he see of a user's world in terms of operations that he knows he can get a computer to perform.

Note that the user cannot normally expect to exploit the full potential of tile computer, but only the potential of the programmer's knowledge of the computer. The need for the programmer as interpreter is a symptom of the sheer difficulty of getting computers to do things, of the current primitive state of software technology. How much better if computers could be made to Interpret for themselves and understand people who want to use them, without programmers.

PROBLEMS WITH CURRENT TECHNOLOGY

What is wrong with current software technology? Essentially, software consists of instructions telling computers how to execute machine operations. Sets of instructions have to be complete before a computer can do anything for you. This pattern occurs at all levels of software, from basic machine operating systems up to a user's application program. You cannot discuss with a computer what you want it to do, unless the detail content of your discussion has been anticipated and programmed into the computer, and that is no discussion.

This almost universal limitation arises from the need, at present, to achieve a one-to-one mapping between a logical model and its implementation in software. Software, in the form of written code transmitted to a computer, has to match the abstract logical representation of a user's world. The logical model has to anticipate the content of information used by people. Before a computer can respond to any charge in the user's world, it is necessary first to return to the program code and change it - a costly process that inhibits users changing their mind.

As an illustration of this tight relationship between a logical model and its software implementation we can consider the familiar (among programmers) records and pointers data structure.

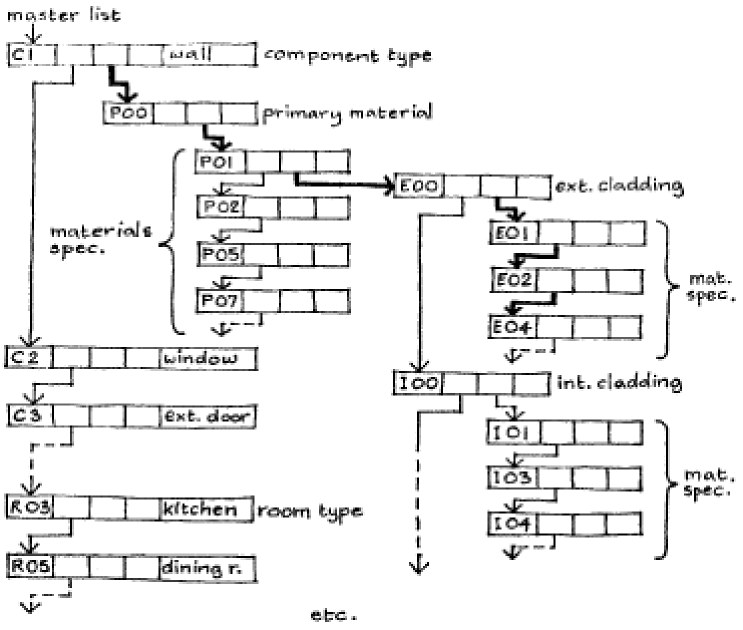


Figure 3: Example of database layout taken from EdCAAD's House Design system (Bijl et al 1971), showing access path to material of external cladding to wall of specified primary material.

Essentially, this requires a decomposition of the subject matter seen in the user's world into its constituent parts. The parts then form records stored in computer memory. The records contain pointers to memory locations of other records. By means of the pointers, assemblies of records can be recreated corresponding to assembled information that is required at different stages of some process, such as a design or analysis process (figure 3).

The crucial issue here is that the structure of pointers and memory locations refers to the physical organisation of computer memory, and this has to be represented explicitly In software. This is why these data structures are commonly described as physical implementations of logical models. The pointers then are the access paths to data stored In memory locations; in any hierarchical arrangement of pointers, only those data that lie on access paths can be accessed and, for any computing application, the appropriate arrangement of pointers has to be complete. The view that a user can get of the stored information and the operations that can be performed on the information is then determined by the data structure.

A skilful programmer will search the user's world to find a durable systematic understanding of users tasks, on which to develop a data structure that Is likely to support mail different instances of task. So the programmer will attempt to postpone the need for reprogramming. Even in the best cases, and especially where programs are offered to designers, sooner rather than later users will want to get at information that may well be stored in the computer but for which there is no programmed access path.

COMPUTERS THAT UNDERSTAND

What can we expect of new software technology? We have now to return to the question of who or what does the interpreting between users and computers. We need to remove our focus from applications programs, that is programs which perform tasks that visibly correspond to the immediate and particular things people want doing. If we focus on what other things computers might do, we can then consider software developments that are aimed at getting computers to understand what people are saying, what they are asking for?

The idea that computers should be. able to interpret demands coming directly from users, and respond appropriately, is not so improbable. We already have experience of communications systems in which the content of communications is conveyed in a manner that is subject to defined and widely accepted rules, where these rules are essential to the interpretation of content and where the rules are not thought to restrict the content. The obvious example is grammatical rules applied to written text. The idea that computers should be able to understand people is not improbable, but its realisation is a long way off. Can we use this promise?

LOGIC PROGRAMMING

We have discussed the one-to-one relationship between logical models and software implementations. If we had a logic system that was implemented in software and which would allow us to formulate different logical models, so that implementation of the models would be an automatic consequence of using the logic system, we would then have a development that would make it easier to change logical models and, in turn, to have computing systems that would be more responsive to changing user demands. The logic system would consist of rules governing the way in which a logical model may be expressed, without limiting the content of the model or the real world application of the model. One implementation of logic in software already exists, in the form of the Prolog logic programming language (Kowalski 1979, Clocksin and Mellish 1981).

Returning to our earlier example of data structures, Prolog allows us to develop a logical model of buildings which employs the equivalent of records and pointers, in the sense of a decomposition of whole objects into parts that have attributes which point to other parts (Steel and Szalapaj 1983). But by using Prolog we do not need to make a prior decision on the totality of a decomposition; parts can be created and related to any other parts at will. This is possible because there is no software implementation of an explicit pointers structure. Connections between paths are made by the expressions used to describe the paths, and a connection exists only for as long as its purpose is being realised. During a dialogue between the user and the model, structures are invoked and dissolved as the user varies the expressions that describe some thing.

Thus if we consider figure 3; we may get the expression:

```
wall <- [primary_material=brick,internal_cladding=plaster]
```

Using generalised knowledge of the construction of such expression, a computer can understand what an expression is meant to convey. It will know that I am talking about a thing that I call a wall which is described by the attributes of primary material and internal cladding which in this Instance are brick and plaster. I can add, modify and delete attributes and their properties as my perception of the wall changes, and I can use my description of the wall as a part of other things that constitute a whole building. Later, I can recall what internal cladding has been specified for the wall by describing the access path in the expression:

```
wall:  Internal_cladding
```

to get the answer, plaster. I can do all this without a programmer having anticipated that I am going to talk about relationships between walls, bricks and plaster.

What we have here is a separation between thinking about some thing and describing what we are thinking to a computer, on the one hand, and the Implementation of our thoughts as software that represents our thoughts and our requests for action in a computer, on the other. The Prolog logic system is responsible for interpreting expressions from a user and, being itself a program, its task is to implement the user's logical model in software. Prolog thus is a general purpose language for conveying intended meaning as well as instructions to computers. Of course, to use the language users must know the rules by which expressions in the language may be constructed.

Prolog is the first implementation of a logic programming language and we should expect further developments of this kind. Such developments offer important advantages for designers, promising new software technology that:

a) logical models. does not demand prior commitment to some explicit model that anticipates the actual content of objects descriptions;

b) partial models: permits incomplete models that represent the state of a user's thinking, allowing subsequent and unforeseen additions to a model;

c) changing models. permits models to be easily modified as the user's perception of something charger., without the need for programmers making corresponding charges to program code;

d) correct models: reduces the need to be right in a predictive sense, models may change as experience shows you were wrong, you may change your mind.

LINE BETWEEN NOW AND THEN

How do we know when we are on the right path, so that we can expect to realise advantages of new technology? Developments in software technology take a long time, in contrast to computer manufacturing technology, and promises are inherently speculative. Yet architects can choose computing environments in which these developments are taking place, and architects can then expect to influence these developments and benefit from early results. This need not be an either/or choice; the same computing environments can also support immediate and practical computing applications.

When you choose a computing application, packaged as a hardware and software system, you are in fact making choicer, on different aspects of computing that will affect the ease with which you can link yourself to new developments (figure 4).

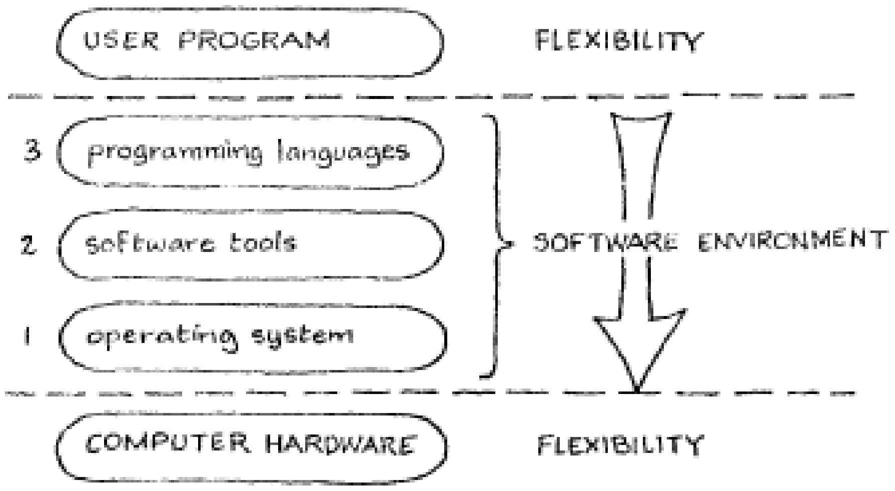


Figure 4: Software environment conditioning the way that computer hardware is made to work for users.

As already discussed, applications programs have to represent what you want to do as instructions that tell a computer how to execute its internal machine operations. Computer hardware is the resource that you wish to exploit.

The software environment governs how you (or your programmer) can exploit computer resources, forming the link between a user program and a computer. As designers, the things you will want to do will charge, and hardware technology changes. Over time, you will want to run different programmes on different hardware. Choice of software environment, made either inadvertently (as an invisible consequence of choosing an applications program) or deliberately, can have major cost implications. Seen by a user, these implications will be experienced as the cost of developing and supporting computer programs, of learning how to exploit program functions, and of making adjustments to a user's existing work-practices to match the anticipations encapsulated in a program. These implications become more severe when it is time to charge a program or to change to new hardware.

A software environment consists of a low level computer operating system plus associated software tools and a rate of high level programming languages. Broadly, choice of operating system is between a computer manufacturer's system or a system developed by computer users. In recent years, there has been a growing preference for the latter, Operating systems that have been developed by and for computer users should offer the following advantages:

a) system visibility: written in a high level language, the traditional barrier between operating system and user programs is diminished and systems can be modified and tuned to suit particular applications fields;

b) portability: systems are not specific to a single manufacturer's range of equipment, and they can be and are ported to different computers, thus easing the job of porting programs;

c) support: given a world-wide user community and given system visibility, these systems have more support resources than can be provided by any single computer manufacturer;

d) efficiency: these systems can get more out of equipment than the manufacturer's own operating system;

e) software tools: as with support, a world wide effort can be and is focussed on software development tools associated with the operating system, to ease the task of writing user programs;

f) programming languages: given the above advantages, people engaged in development of new software technology and new programming languages are likely to be working with these systems, and results will be available first on these systems.

A dominant example is the UNIX [*] operating system. This system is so persuasive that manufacturers of powerful new 16 and 32-bit microcomputers are now supplying their computers only with UNIX, without attempting to develop their own operating systems. The UK programme for advanced Information technology has chosen UNIX as a preferred operating system. The Prolog logic programming language, as a forerunner of new technology, has already been implemented on UNIX (Pereira 1982). At the same time, UNIX is rapidly gaining popularity as an environment for immediate and practical computer applications.

[*] UNIX is a Trademark of AT&T

Thus, taking a cautious view of the simplifications necessarily contained in this paper, it is possible to relate immediate decisions on how to get into computing to the promise of future mew technology. For designers, this promise is important. If current computing technology were to be regarded as being mature and if the world were to became dependent on it, then architects and architecture would face a dismal prospect. If, on the other hand, we can recognise that present computing Is primitive and that the technology is due to develop and charge, and if architects among ordinary people can take part in influencing that change, then we have reason for optimism and excitement.

LESSONS POR ARCHITECTURAL EDUCATION

Drawing on the arguments presented in this paper and summarizing the implications, we can distill the following directives for architectural education.

1 Given a changing technology, we should be less concerned to train students to use existing computing programs - those of us who are responsible for developing programs should be less concerned to impress our achievements on others.

2 It is necessary to differentiate between learning about:
  a) computing technology as a mode of working:
  b) the subject matter to which computers are applied (e.g. building science topics).

3 We should be more concerned to impart to students a fundamental understanding of technology and how it may relate to activities of people.

4 We need to convey ideas on how technology may charge, why it should change and how architects can influence that charge.

5 Hands-on experience of writing or using programs should be viewed as an important means of gaining insight into the nature of current technology, to stimulate idea on how it should be better.

6 We should be attempting to extract principles (and theory?) from our experience to raise the intellectual rigour of our work, as a sound basis for teaching.

7 We should be far more receptive of work from other fields of computing (at the present moment, particularly the field of Artificial Intelligence) and learn from their methodology as well as their products.

8 Our teaching has to have relevance to the next 40 years of a student's working life - we need to distinguish between transient local interests and probable durable aspects of computing technology.

ACKNOWLEDGMENTS

The views presented in this paper have emerged from work at EdCAAD funded by the Science and Engineering Research Council, plus experience of EdCAAD computer programs in use in architects' offices.

In recent years, this work has benefited from staff joining EdCAAD from the Department of Artificial Intelligence at Edinburgh University.

REFERENCES

Alvey Report; "A Programme for Advanced Information Technology", HMSO, 1982.

Bijl, A., Renshaw, A., Barnard, D.F.; "House Design: Application of Computer Graphics to Architectural Practice", EdCAAD report, 1971.

Bijl, A., Stone, D. and Rosenthal, D.S.H.; "Integrated CAAD Systems", EdCAAD, 1979.

Bijl, A.; "Non-Prescriptive Computing Technology for Designers", proc. Design Policy Conference, London, July 1982.

CIAD et al; "The Effective Use of Computers within the Building Industries of the European Community", EC report 1979.

Clocksin, W. and Mellish, c.; "Programming in Prolog", Springer-Verlag, 1981.

Dickson, P.; "The Official Rules", Dellta Books Publishing Company, New York, 1978.

JIPDEC; "Preliminary Report on Study and Research on Fifth-Generation Computers 1979-1980", JIPDEC, 1981.

Kowalski , R. A. ; "Logic for Problem Solving", Elsevier North Holland, 1979.

Pereira, F.; "C-Prolog User's Manual", EdCAAD, 1982.

Steel, S. and Szalapaj, P.; "Pictures without Numbers: Graphical Realisation of Logical Models", PARC83.