

VRIJE UNIVERSITEIT  
AMSTERDAM



SUBFAKULTEIT PSYCHOLOGIE  
VAKGROEP FUNKTIELEER EN METHODENLEER  
DE BOELELAAN 1115      1081 HV AMSTERDAM-BTV.



'COGNITIVE ENGINEERING'

A conference on the psychology of  
problem solving with computers

Vrije Universiteit Amsterdam

10 - 13 Augustus 1982

MINI-PAPERS



This conference  
is  
sponsored by

**CONTROL DATA B.V.**

Postbus 111  
2280 AC Rijswijk Z.H.  
Telefoon (070) 949344  
Telex 33037

COGNITIVE ENGINEERING

A conference on the psychology of problem solving with computers  
Vrije Universiteit Amsterdam  
10 - 13 augustus 1982

Organising committee:

Thomas Green, Sheffield  
Gerrit van der Veer, Amsterdam

Program committee:

Thomas Green, Sheffield  
Jean-Michel Hoc, Paris  
Anker Halms Jørgensen, Copenhagen  
Susanne Maass, Hamburg  
Steve Payne, Sheffield  
Gerrit van der Veer, Amsterdam

Congress secretariat:

Elly Lammers,  
Subfaculteit Psychologie  
Vrije Universiteit  
De Boelelaan 1115, Prov.I.B.112  
1081 HV Amsterdam  
The Netherlands  
Tel.: + 31 20 548 3870/3869

Registration:

Main Hall, main building (Hoofdgebouw)  
Vrije Universiteit  
De Boelelaan 1105  
Amsterdam

Paper session:

Universiteitsraadzaal, Hoofdgebouw  
Vrije Universiteit  
De Boelelaan 1105  
Amsterdam.

Table of contents:

<u>Timetable</u>		8	
<u>I. The psychology of the computer user, a. general</u>		11	
N. Hammond, A. MacLean, P. Barnard	Knowledge fragments and users' models of the system.	13	
K. Bø	Human interaction with graphics systems	15	
J. Preece	Graphs are not straightforward	29	? >
S.J. Payne	The perception of grammars: higher order rules	41	? <<
M.J. Tauber	Psychology of programming, - general aspects and aspects of teaching and learning of programming language.		< ')
E. Nullmeier/ K. Roediger	Developing an instrument for the analysis of cognitive activities of workers at dialogue systems	53	<>
<u>I. The psychology of the computer user, b. the programmer</u>		65	
G.C. v.d.Veer, J.v.d.Wolde	Individual differences and aspects of control flow notation	67	? .
H.E. Sengler	A model of the programmer's abilities to understand program semantics and its impact on program(ming language) design.	77	? .
J.M. Hoc	Analysis of beginners' problem-solving strategies in programming.	85	◇
J.H. Kahney	Problem solving by novice programmers	95	◇
R.P. v.d. Riet	Measuring the performance of students in an introductory informatics course.	111	<<
W. Volpert, R. Frommann	Software assessment from the viewpoint of the psychology of action.	133	? <<
P. Naur	Program development studies based on diaries	135	? <<
H.D. Böcker, G. Fischer, R. Gunzenhäuser	Project INFORM: The function of integrated information manipulation systems (IMS) to support man-machine-communication	143	? <<<
<u>II. Facilitating human-computer interaction, a. Tools and aids</u>		145	
B. Senach	Computer aided decision making with graphical display of information	147	
S. Hägglund	The case for control independence in dialogue-oriented software.	155	? <<
T.R.G. Green	Display of program structure - why and how?	165	? <<
A. Arblaster	The evaluation of a programming support environment	171	? <<

' ) abstract not available at time of reproduction

Table of contents:

Timetable

II. Facilitating human-computer interaction, b. system design issues 175

S. Maass Why systems transparency? 177

L. Pinsky What kind of "dialogue" is it when working with a computer? 185 ?

A.H. Jørgensen Naming commands: an analysis of designers naming behaviour 1) ?

A. Dirkzwager Inside and outside the system, problems of communication, interaction and understanding in a programmable environment. 195 K<

---

1) abstract not available at time of reproduction





Timetable

Tuesday 10 august

13.00 - 17.00 registration  
exhibition of books on computers & psychology

Wednesday 11 august

Opening session, chairman G.C. van der Veer (Amsterdam)  
10.00 Prof. Dr. H. Verheul, Opening of the conference  
Dean of the Vrije Universiteit  
10.30 Coffee

I. The psychology of the computer user

a. General, chairman M. Zoepprits (Heidelberg)

11.00 N. Hammond, A. MacLean, Knowledge fragments and users' models of the system.  
P. Barnard (Cambridge)  
11.45 K. Bø (Trondheim) Human interaction with graphics systems  
12.30 Lunch  
13.30 J. Preece (Milton Keynes) Graphs are not straightforward  
14.15 S.J. Payne (Sheffield) The perception of grammars: higher order rules ←  
15.00 Tea  
15.30 M.J. Tauber (Paderborn) Psychology of programming, - general aspects and aspects of teaching and learning of programming language.  
16.15 E. Nullmeier/K. Roediger Developing an instrument for the analysis of cognitive activities of workers at dialogue systems.  
(Berlin)  
17.00 Reception offered by the University

Thursday 12 august

I. The psychology of the computer user

b. The programmer, chairman J.M. Hoc (Paris)

9.00 G.C.v.d.Veer (Amsterdam) Individual differences and aspects of control flow notation.  
J.v.d. Wolde (Twente)  
9.45 H.E. Sengler (Hamburg) A model of the programmer's abilities to understand program semantics and its impact on program(ming language) design.  
10.30 Coffee  
11.00 J.M. Hoc (Paris) Analysis of beginners' problem-solving strategies in programming.  
11.45 J.H. Kahney (Milton Keynes) Problem solving by novice programmers.  
12.30 Lunch  
13.30 R.P. van de Riet (Amsterdam) Measuring the performance of students in an introductory informatics course. ←  
14.15 W.Volpert/R.Frommann (Berlin) Software assessment from the viewpoint of the psychology of action. ←  
15.00 Tea  
15.30 P. Naur (Copenhagen) Program development studies based on diaries. ←  
16.15 H.D.Böcker/G. Fischer/ Project INFORM: The function of integrated information manipulation systems (IMS) to support man-machine-communication ←  
R. Gunzenhäuser (Stuttgart)

Conference dinner  
(time to be announced)

Friday 13 august

II. Facilitating human-computer interaction

a. Tools and aids, chairman E. Edmonds,

9.00 B. Senach (Le Chesnay) Computer aided decision making with graphical display of information

9.45 S. Hägglund (Linköping) The case for control independence in dialogue-oriented software. <<

10.30 Coffee

11.00 T.R.G. Green (Sheffield) Display of program structure - why and how? <<

11.45 A. Arblaster (London) The evaluation of a programming support environment <<

12.30 Lunch

b. System design issues, chairman F. van der Ham, (Amsterdam)

13.30 S. Maass (Hamburg) Why systems transparency?

14.15 L. Pinsky (Paris) What kind of "dialogue" is it when working with a computer?

15.00 Tea

15.30 A.H. Jørgensen (Copenhagen) Naming commands: an analysis of designers' naming behaviour <<

16.15 A. Dirkzwager (Amsterdam) Inside and outside the system, problems of communication, interaction and understanding in a programmable environment. <<

17.00 Closing session, chairman T.R.G. Green, (Sheffield)  
Prof. J.M. van Gorscot, (Amsterdam)



I. THE PSYCHOLOGY OF THE COMPUTER USER

a. General



KNOWLEDGE FRAGMENTS AND USERS' MODELS OF THE SYSTEM

Nick Hammond, Allan MacLean & Philip Barnard  
MRC Applied Psychology Unit, Cambridge

To learn and use a complex interactive system efficiently, users have to structure the information concerning the system. This information includes the user manual and any help facilities in addition to details of the interface and task. The better the structuring, the more usable the system. While the ideal may be for the user to have a unified model of the whole system, this rarely seems to be achieved in practice for several reasons. First, it may be impossible to create a single metaphor adequate for dealing even with a section of any real system (Halasz & Moran, 1982). Second, the information structures generated by users in the course of learning may typically not be unitary: even when a good metaphor is available, the user's concept of the system is usually fragmented, with each fragment having a restricted sphere of applicability.

We illustrate this aspect of users' information structures through examples taken from studies of user difficulties in three contrasting interactive systems. One of these is reported in Hammond, Long, Clark, Barnard & Morton (1980). Several general themes emerge. First, users tend to form partial or fragmentary information structures to explain and predict local aspects of the system. One knowledge fragment may be quite independent of or even inconsistent with another fragment if they address the same area. For instance, in one system users conceptualised the screen either as a tabula rasa on which relevant material could be written or removed at will, or as an information display controlled by the system and modified only via indirect commands. Contexts in which these roles were mixed caused difficulties. Second, users do not always successfully access at the appropriate time such knowledge fragments as do exist, although they may quickly realise their error: "I'm always doing that" is a common enough response to a system error message. Third, particularly during the early stages of learning, users seem to use very little evidence before forming an explanatory fragment (Lewis & Mack, 1982). However, once a fragment which successfully predicts some aspect of system behaviour has been formed, users are loth to reject or even modify it. Thus in one system, the screen cursor could be moved either by MOVE Keys (to an adjacent character cell) or by TAB Keys (to an adjacent input field). Users who learned through the training manual about these keys in the context of a menu with only a single input

field found that the TAB Keys had no observable effect while the MOVE Keys behaved as expected. Typically, from this single learning trial, such users never attempted to use the TAB Keys again in a further two days of training.

These themes have consequences both for the learning and for the use of systems. Early examples are especially potent for forming user knowledge fragments, both appropriate and inappropriate, and should therefore be chosen with discretion. Following initial learning, the user can be helped in accessing the suitable fragment by the provision of signposts, clues to relevant aspects of the local task and system contexts, embodied in display headings and layout, in documentation and in error messages. The presentation of the system should both encourage the creation of a well-structured set of knowledge fragments and provide signposts to their use.

#### References

- Halasz, F. & Moran, T.P. (1982). Analogy considered harmful. Proceedings of Human Factors in Computer Systems, pp 383 - 386.
- Hammond, N.V., Long, J.B., Clark, I.A., Barnard, P.J. & Morton, J. (1980). Documenting human-computer mismatch in interactive systems. Proceedings of 9th International Symposium on Human Factors in Telecommunication, pp 17 - 24.
- Lewis, C. & Mack, R. (1982). Learning to use a text processing system: Evidence from "thinking aloud" protocols. Proceedings of Human Factors in Computer Systems, pp 387 - 392.



HUMAN INTERACTION  
WITH GRAPHICS SYSTEMS

by

Ketil Bö  
Stiftelsen for industriutvikling  
P.O.Box 660  
N-7001 Trondheim  
NORWAY

## INTRODUCTION

The graphics presentation techniques must be adapted to the human peculiarities in order to convey the desired messages as accurate and natural as possible from the computer to the user.

These messages may range from pure information represented by a single value to pure esthetic, represented by an artistic picture. To make this possible, the users perceptual and cognitive processes must be taken into consideration during the development of the graphics presentation techniques.

### 1. THE HUMAN EYE

The human eye is a radiant energy receiver, sensitive to the 0.4 to 0.7  $\mu\text{m}$  band of the spectrum.

The human focuses the image of any object onto the retina by means of the lenses. On the retina is a complex mosaic of two types of receptors, the rods and cones. The rods are generally sensitive to light. The cones, which are less sensitive, contain absorbing filters for blue, green and red and therefore give the colour information to the brain.

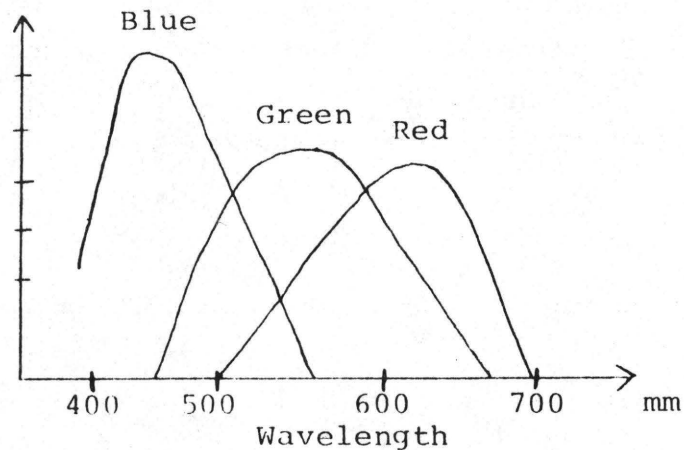


Fig. 1. The eyes pigment absorption curve.

There are, however, experiments that indicates that the rays are not colour-making in themselves. Rather they are bearers of information that the eyes uses to assign appropriate colours to various objects in an image. It seems that the eye perceives colour by comparing longer and shorter wavelength and the experiments shows that 588 nm is the balance point between short and long wavelengths in this context [1].

The receptors in the eye contain an organic substance, rhodopsin. Incoming radiant energy causes a chemical reversible transformation in this material which generates electronical signals that are transmitted to the brain by fibers of the optic nerve.

This transformation of rhodopsin has its own build up time and decay time and acts as a low pass filter which explains why the eye accepts the pulsating beam of the CRT as long as the refresh rate does not drop below the flicker frequency limit.

The fibers of the optical nerve can be divided into two categories; fibers that are only sensitive to luminance and fibers that are only sensitive to colours. To exploit fully the capabilities of the human vision, the information should therefore be given in colours.

The retinal image is transformed into a visual cortex of the brain by means of a complex set of processes which we are now beginning to understand.

### 1.1 The constancy factor

One of those processes is known as the constancy effect which makes a dark grey area look dark grey in all sorts of light. The reason for that effect is that the perceived grey scale is dependent on the ratio between the light intensities of the particular area and the light intensities reflected from adjacent regions.

This is demonstrated by a constant grey disk with a changing ring around. The shade of the disk seems to change from light to medium to dark grey as the ring light is first made twice as intense, then four and eight times as intense than the light of the disk.

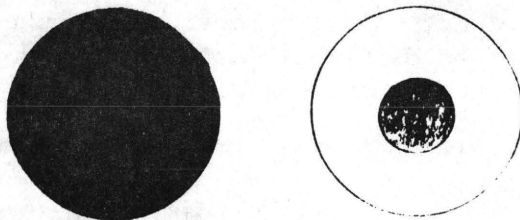


Fig. 2. Constant grey disk with changing ring.

The same goes for colour: a bright yellow disk change to dark brown when surrounded by a ring of high intensity white light.

### 1.2 The contours

Another important feature in the visual process is the contours. Contours are so dominant in our visual perception that when we draw an object, it is almost instinctive that we start by sketching its outline. We see contours when there is a contrast, or difference, in the brightness or colour between adjacent areas. In fact the visual system tends to abstract and accentuate the contours in patterns of varying contrast. This phenomena called the Mach bands is seen as a narrow dark band at the dark edge and a narrow bright band at the bright edge.

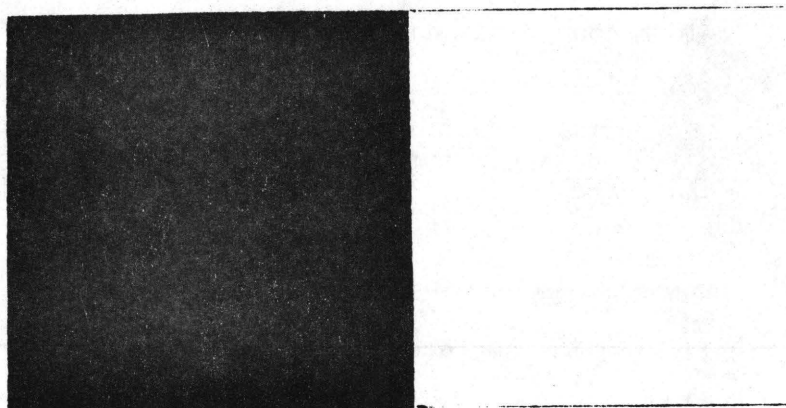


Fig. 3. Mach band in a contour.

It seems to be an universal principle that contours are enhanced and uniform areas are lost. (Ex. copying of a uniform black area.) Another interesting contour effect can be demonstrated by separating two identical grey areas with a special contour that has a narrow bright spur and a narrow dark spur. Although the two uniform areas away from the contour have same objective luminance, the gray of the area adjacent to the light spur appears to be lighter than the grey area adjacent to the dark spur. This demonstration shows that the contours have effect also on the objective uneffected areas of the images. [2]

### 1.3 Perception of transparency

We may distinct between physical and perceptual transparency, because they are not always found together.

Transparency is perceived when one sees not only surfaces behind a transparent medium, but also the transparent medium itself. (Pure air is not perceived as transparent.) On the other hand it is possible to generate the illusion of transparency even if it is not physical.

What causes then perceptual transparency? As with other visual phenomena, the causes must be sought in the pattern of stimulation and in the processes of the nervous system resulting from the retinal stimulation.

Some of the conditions for perceiving transparency is known:

First of all perceptual transparency depends on the spatial and intensity relation of light reflected from a local area.

If the light reflected from two different colours reaches the same retinal region, an intermediate colour will generally be perceived. It turns out, however, that if both the transparent object and the background object are perceived as independent objects, both colours can be seen. This phenomena is called colour splitting and works in a direction opposite to the well known colour fusion.

The proportion of the stimulus colour going to each of the perceived surfaces can be described by the formula:

$$p = \alpha a + (1 - \alpha)t$$

where  $\alpha$  stands for the proportion of the colour in the opaque layer a. The remainder of the colour goes to the transparent layer t.

There are both special colour conditions and figural conditions for perceiving transparency.

The colour condition can be derived from the given formula. The main spatial conditions for perceiving transparency are:

- Figural unity of the transparent layer.
- Continuity of the boundary line of the underlying layer (visible).
- Adequate stratification of the surfaces. [3]

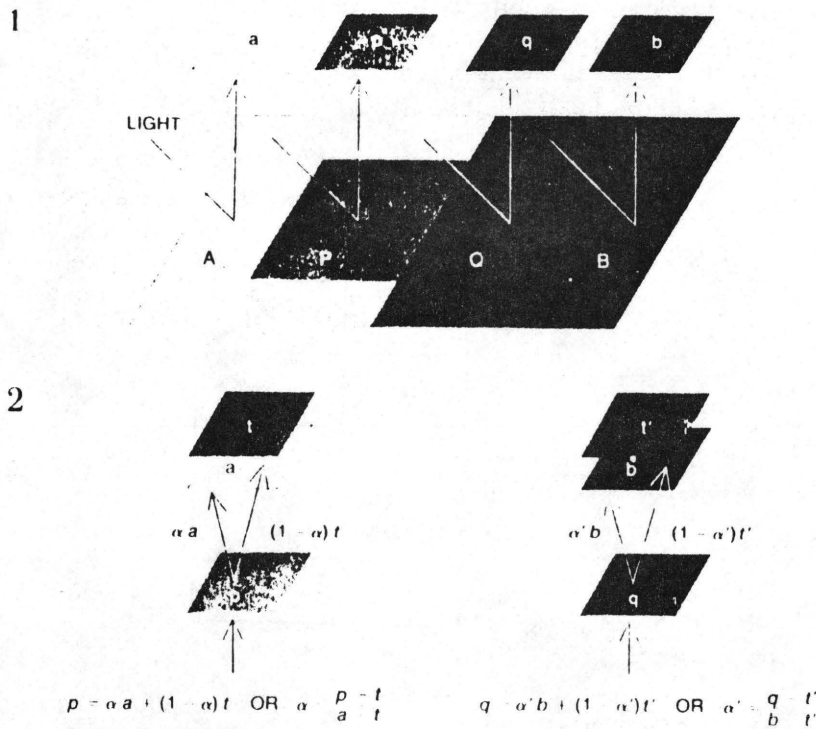


Fig. 4. Colour splitting in connection with transparency.

#### 1.4 Perception of Shapes

A graphical image of a real object is a two dimensional representation built up by primitive elements such as points, lines and polygons.

This representation must be presented in such a way that the observer perceives a true model of the real object.

To achieve this, the human capabilities of perceiving shapes must be understood and utilized.

The visual processing by the brain begins in the lateral geniculate body which continues the analysis made by the retinal cells. In the cortex "simple" cells respond strongly to line stimuli, provided that the position and orientation of the line are suitable for a particular cell. The visual cortex rearranges therefore the input in a way that makes lines and contours the most important stimuli.

Movement is similarly handled by specific cells which are most sensitive to particular rates and directions, and is therefore also an important stimulus factor for the visual processes. The further organization of the visual information is closely studied by the gestalt psychologists which has generalized their results into five organizational principles [4]:

1. The perceptual field is structured into figure and ground.

Typically we see the figure in greater detail than the ground and perceive it as being in front.

Another important aspect of the figure-ground relation is that it can be reorganized spontaneously and very rapidly.

Perception of other relations are also subject to sudden reorganization.

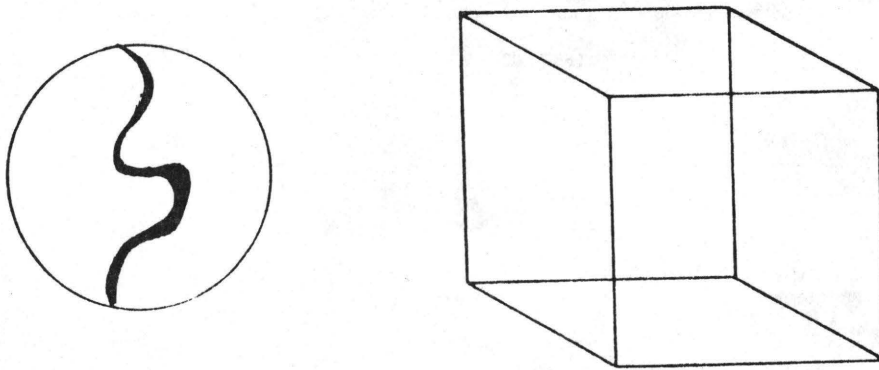


Fig. 5. Ambiguous figures.

2. The grouping of elements into wholes depends on the properties of the elements and their arrangements.

- i) Grouping depends on proximity
- ii) Grouping depends on similarities
- iii) Grouping depends on good continuation
- iv) Grouping depends on closure

3. Organized wholes tend towards closure, simplicity, symmetry and regularity.

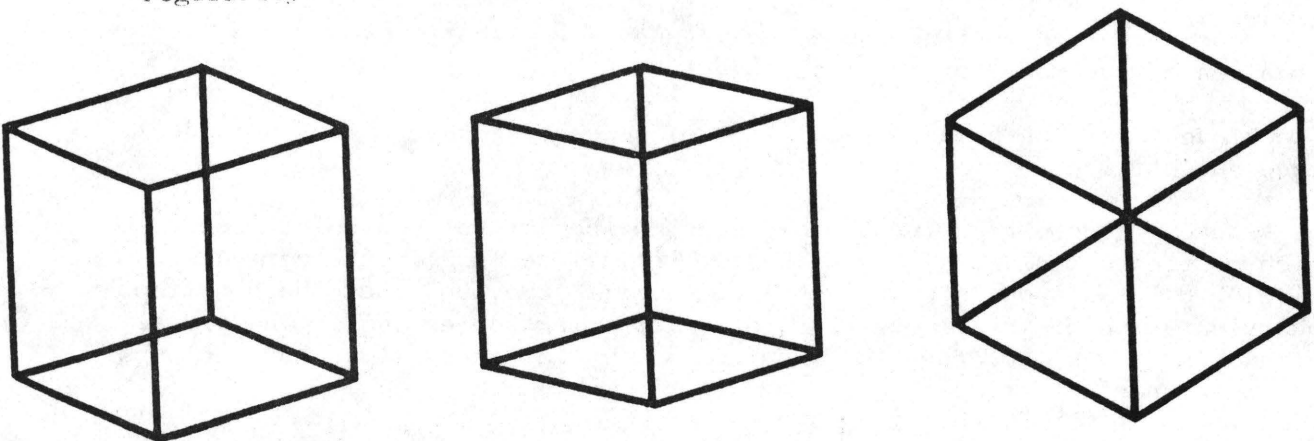


Fig. 6. Tristable shapes (The Necker cube).

4. Organized wholes tends to be perceived as objects.
5. Objects are perceived as having constant properties.

We perceive an rotating object moving away from us in changing lighting conditions as having constant shape, size and colours.

### 1.5 Depth perception

Another important factor for Computer graphics is the visual mechanisms for perceiving depth from the retinal images which is pure two dimensional. It is possible to judge distance from a visual image by using indirect cues such as the angle subtended by an object of known size, the effort of focusing the lense of the eye or the effect of motion parallax, but these cues can not be used in all situations and they are not as accurate or immediate as the powerful sensation of depth coming from the binocular vision system.

The binocular vision system split the impulses from the right part of the two retinas to the right side of the visual part of the brain and the impulses from the left halves to the left side of the visual part of the brain.

Utilizing the binocular parallax the distance to the object is predicted.

### 1.6 Perceived movement

As we have seen, the visual system has its own specialized cells for detecting direction and rates of movement. From the experience with films etc. we know that it is possible to perceive movements from a serie of pictures shown with close intervals. But what is the relation between the perceived movement and time intervals which create the illusion?

To answer this an experiment was performed by Wertheimer [5] which used a pair of lines each presented very briefly (about 50 ms) in rapid succession. The time interval between "on" and "off" time for the lines was then varied:

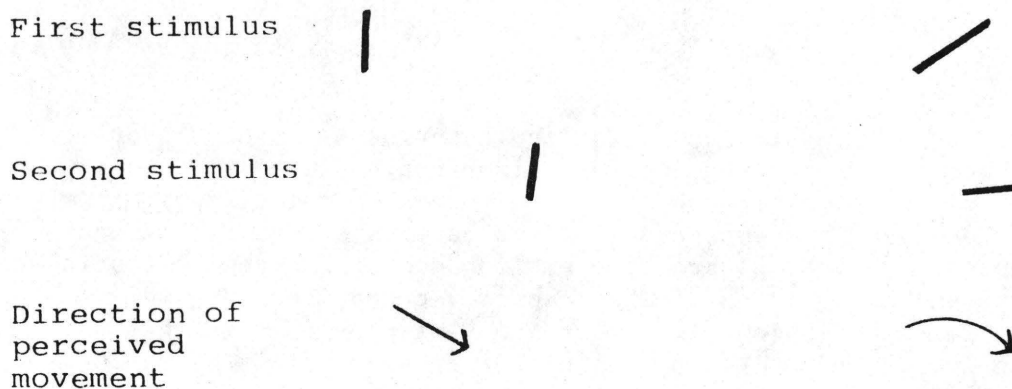


Fig. 7. Perceived movement.

When the interval between the two exposures was less than 25 msec., the two lines was seen as being simultaneously present with no motion at all. When the interval was longer than 200 msec., the two lines was seen as appearing successively, but without any movement between them. The illusion occurs when the time interval is manipulated in the lower part of the range, between about 25 and 400 milliseconds. As the interval is increased gradually from 25 milliseconds, the appearance of simultaneity gives way to that of movement. The first line appears to move part of the distance toward the second and then disappears. Then the second line appears, displaced toward the first, and moves toward its own actual location.

With further increases in the time interval between the two flashes, the distance between the disappearance of the first line and the appearance of the second diminishes until the observer perceives what is called optimal movement: a single line moving smoothly and continuously across the space from its origin to its terminus.

As the time interval between flashes is increased still further, optimal movement continues to be seen, but its speed grows successively slower. Finally one no longer sees an object moving across the space. Instead most observers have a sense of movement itself: objectless, "pure" movement, which Wertheimer calls "phi movement". The last stage of the perception is a slow sequence of flashes as first one and then the other line comes on and goes off. To sum up: Small variations in the interval of time between the two flashes produce five distinctly different perceptions: simultaneity, partial movement, optimal movement, phi movement and succession.

Apparent movement, seems to relate to two mechanisms in the human perception. One is the amount of time the visual system requires to form the perception of simple figures. The other is impletion.

From many experiments we know that the visual system takes about 300 milliseconds to form the perception of a simple object one is prepared to see.

In other words, one can form certain visual perceptions at an optimum rate of about three per second. Many kinds of simple stimuli that occur within 300 milliseconds, which is to say at faster than the optimum rate, can interact so that the occurrence of each affects the appearance of the other. For example, sequences of letters presented at such a rate can be confused by the observer. He may detect the letters but confuse their sequence. Thus there is something special to the human visual system about rates in the region of three cycles per second. One of the results is the illusion of movement. [5]

### 1.7 Visual illusion

Discrepancies between object and image are particular evident in the distortions of size and shape that occur when certain figures are viewed. These illusions may arise from our visual mechanisms that under normal circumstances made the visible world easier to comprehend. These particular manifestations of the visual processes may cause severe problems for computer graphics leading to misinterpretations and ambiguities if the necessary precautions are not taken. The simplest illusion, which also cause a lot of problems, is that vertical lines looks longer than horizontal lines of equal length. No satisfactory explanation is found for this phenomena, but all evidence suggest that the distortions originate in the brain and not in the eye.

The most famous of the distortion illusions is the double headed arrows:



Fig. 8. The arrow illusion.



Here the arrow with the arrow-head pointing outwards looks shorter than the one with the arrow-heads pointing inwards.

This illusion is equal to the eyes tendency to expand inside corners of a room and shrink the outside corners of the structures. The illusion may occur because the arrows resembles outline drawings of corners seen in perspective.

Another perspective illusion is the railway line illusion where the further line looks longer than the closer line of same size.

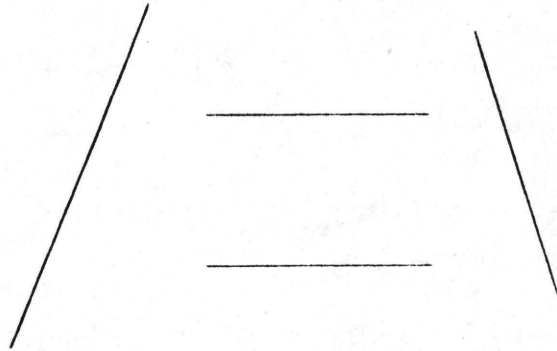


Fig. 9. The railway illusion.

We also have the multistable images which is spontaneous changing the appearance as we look steadily at them. We have already mentioned the figure-ground pictures. Best known is the Necker cube, which is a line drawing of a transparent cube.

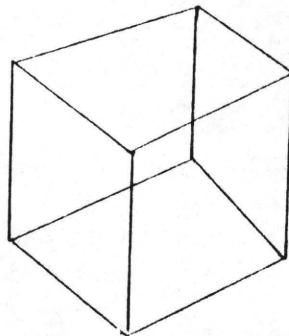


Fig. 10. The tristable Necker Cube.

If one looks steadily at the cube for a while, it will suddenly reverse the depth. What was the back face now is in the front. The two orientations will alternate spontaneously, sometimes one is seen, sometimes the other, but never both simultaneously. This cube is actual tristable because it may also be seen as a two dimensional figure.

This example shows very clear the problems connected with representing three dimensional objects in two dimensions. In a sense all pictures are "impossible" because they have a dual reality. They are seen both as patterns of lines laying on a drawing surface and as an object depicted in a quite different three dimensional space. Viewed as patterns they are seen as being two dimensional. Viewed as representing other objects, they are seen in a quasi-

three-dimensional space. The pictures are also ambiguous, because the third dimension is never precisely defined. In the Necker cube this ambiguity is so great that the brain never comes up with a single interpretation. [6]

## 2. COGNITIVE PROCESSES

Other important parameters when graphics presentation is concerned, is how a human is encoding, storing and retrieving information.

This is particular important for deciding on screen lay-out, prompt, messages and command handling.

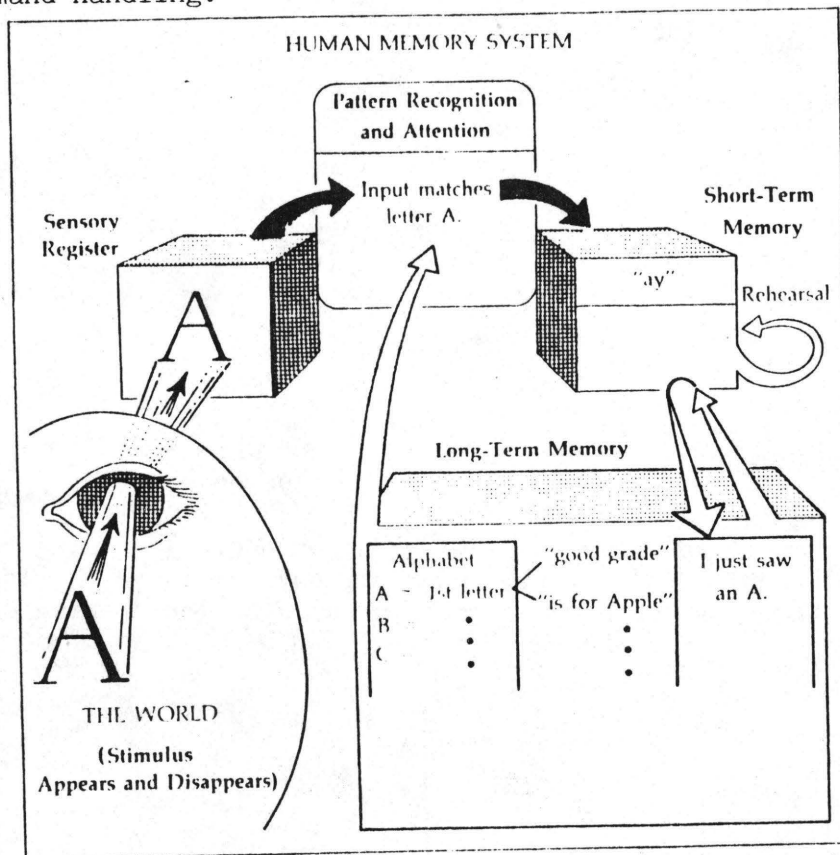


Fig. 11. A model of the human information processing system.

### 2.1 The human information processing system

The sensory register is a very short term register (for vision less than 1/3 of a second) which serves the function of briefly holding the "raw" information of a stimulus until it can be erased or transformed into a new form and sent further into the system. While the information stays in the register, two important processes come into play:

1. Pattern recognition, which result in contact between the information in the sensory register and previously acquired knowledge. The result of this process is "labelling" of the information. That means to convert raw information into something meaningful.
2. Attention, which makes it possible to focus on the relevant information and filtering out the rest.

Input to the system that have been recognized and attended is passed to the

short-term memory (STM). The details of the short-term memory is still not fully understood, but there are strong evidences that a labelled item will stay only a short time in STM if it is not rehearsed. The capacity of the short-term memory seems to lay in the neighbourhood of seven unrelated items. Information can, however, be grouped together and kept in the STM occupying only one "space".

Therefore, we are justified in assuming that our memory are limited by the number of unrelated units or symbols we can master, and not by the amount of information that these symbols represent. Thus it is helpful to organize our information intelligently and take full advantages of this in our design of the human/computer communication process.

Rehearsel, however, can be used to keep the information in the short-term memory. It seems also to strengthen the representation of the information in the long-term memory in such way that it is easier to recall later.

Another interesting finding is that a verbal label held in STM is coded acoustically. That means it is easier to mix information that sounds equal, than information that looks equal.

Finally selected information is sent into an essentially permanent storehouse called the long-term memory (LTM), which holds all our knowledge about the world. The information in LTM seems to be coded in many ways acoustically (we recognize sounds), visually (we recognize pictures) or semantically (we recognize meanings).

## 2.2 Recall and recognition

Our everyday experiences tells us that recognition is usually easier than recall. Only in exceptional circumstances these relationships may be reversed. We may for instance recall correctly the spelling of a word, only to fail to recognize that it is correct.

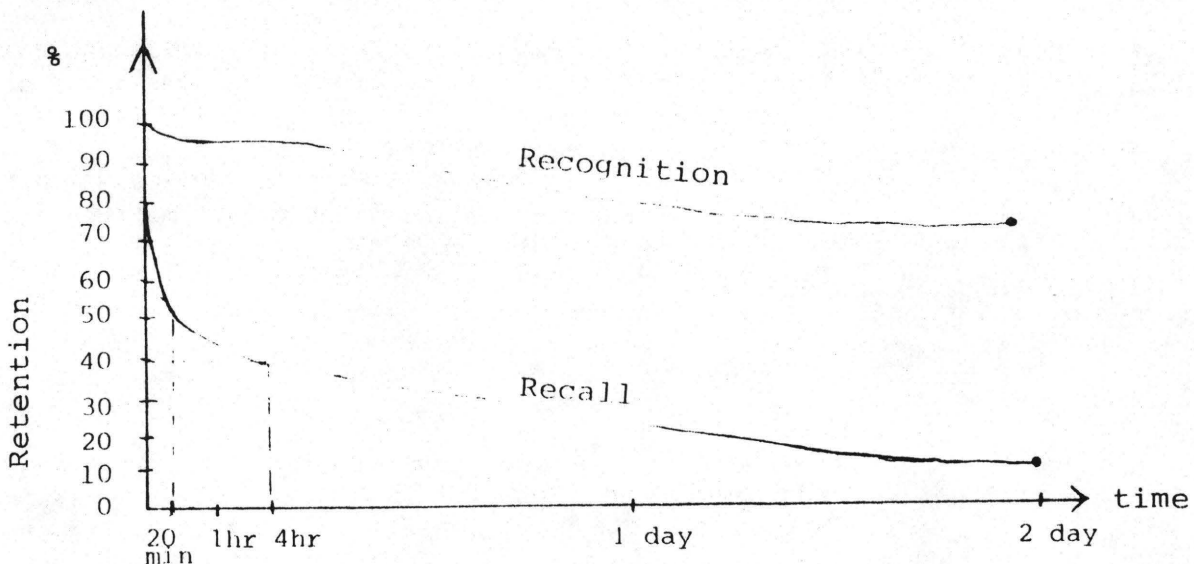


Fig. 12. Comparison of recognition and recall.

Experiments on words, sentences and pictures show that recognition performance is extremely high, relative to recall.

Shepard (1967) performed a test with subjects seeing 612 coloured pictures, their recognition accuracy was 97% when given a two alternative forced-choice test.

Another fact about recognition testing is that performance remains high even with long retention intervals (up to 120 days). [7]

### 3. USER PERCEPTUAL AND COGNITIVE SUMMARY

We have discussed a number of human perceptual and cognitive processes which is important for the understanding and improvement of the graphics presentation techniques:

- The human eye is directly sensitive to the blue, green and red part of the colour spectrum and the optic nerve has special fibers for transmitting colour information to the brain.
- The chemical processes on the retina has its own build up and decay time, which acts as a low pass filter which makes the refreshing of the CRT acceptable to the eye as long as it is not falling below the flicker frequency limit.
- The constancy effect makes it easier to comprehend the information on the graphics surface independent of intensity, shape and size of the objects.
- The contour effect forces concentration on the contour rather than on the interior of objects, but this effect also strengthen the staircase effect on the raster devices.
- Perceptual transparency makes it possible to generate transparent objects on a graphics screen device.
- Utilizing the five organizational principles developed by the Gestalt psychologists, makes it possible to use relatively simple pictures and still convey the message.
- By the effect of perceived movement, controlled animation can be achieved.
- Visual illusions may cause problems for computer graphics if the necessary precautions are not taken.
- The sensory registers keeps the raw information less than 1/3 sec.
- The short-term memory is limited to about seven independent information units and without rehearsal the information will stay only for a short period.
- Information may be grouped and named, thus occupying only one space in STM.
- Recognition is far more efficient than recall.

REFERENCES

- [1] Land, H.E.: "Experiments in colour version". Readings from Scientific American
- [2] Ratliff, F.: "Contour and contrast". Image, Object and Illusion, Readings from Scientific American 1974
- [3] Metelli, F.: "The Perception of Transparency". Image, object and Illusion, Readings from Scientific American 1974
- [4] Hayes, J.R.: "Cognitive Psychology. Thinking and creating". The Dorsey Press 1978
- [5] Kolers, Paul A.: "The illusion of Movement". Reading from Scientific American 1964
- [6] Gregory, Richard L.: "Visual Illusion". Image, Object and Illusion, Readings from Scientific American 1974
- [7] Klatzky, Roberta L.: "Human Memory, Structures and Processes". W.H. Freeman and Co. 1975



Title:            Graphs are not straightforward

Author:           Jenny Preece  
Centre for Continuing Education  
Open University  
Milton Keynes MK7 6AA  
U.K.

Submitted to:    Cognitive Engineering  
Amsterdam, 10-13 August 1982

Abstract

In this paper I shall discuss the kinds of errors that pupils make in interpreting cartesian graphs. The data upon which the discussion is based comes from two tests.

One of the tests involves interpreting a multiple curve graph, in which the curves are independent, whilst in the other test the curves are dependent. Both of these kinds of graphs are frequently used to display data from computer simulation programs.

Finally, I shall suggest some guidelines which will help authors of computer assisted learning (CAL) programs to design and use graphs more effectively.

GRAPHS ARE NOT STRAIGHT FORWARD

Introduction

Many extraordinary and unsubstantiated claims are made about the educational potential of computer generated graphs. However, little real attention has been given to the fundamental concepts involved in graph interpretation. The ways that students deal with these concepts must be identified and explained before displays can be designed which will capitalise on the potential offered by the interaction, speed of delivery and colour available in today's microcomputers.

In this paper I will present some empirical evidence which illustrates the kinds of errors that pupils make in interpreting cartesian graphs.



1. Multiple curve graphs

Two kinds of multiple curve graphs are frequently used to display data from computer simulations. The simplest kind of graph has curves representing independent variables which pupils have to compare. Sometimes the variables will be different. More usually the curves will represent the same variable which has been subjected to different experimental conditions in the simulation (e.g. the effect of several temperatures on an enzyme reaction).

In the other kind of multiple curve graphs the variables represented by the curves are not independent. Consider, for example, figure 1 which was produced by a computer simulation program called POND (Leveridge 1978) designed for pupils of 15-18 years of age. Pupils investigate how the population levels of the three kinds of organisms are affected by their own initial population sizes (figure 11), by fishing and by pollution. The pupils are meant to interpret the graphs, form hypotheses, test their hypotheses by setting parameters in the program, examine the results, and reassess and refine their hypotheses etc. The variables are inter-dependent, so pupils have to identify and explain cause and effect relationships.

Pencil and paper experiments have been done to isolate the kinds of errors that pupils make when there is no interaction with the computer. Ways of more effectively using the unique features of the microcomputer will be investigated, based upon this work, in a future study.

The experiments described in sections 2 and 3 were performed with 14 and 15 year olds. Empirical evidence suggests however, that many adults make the same kinds of errors.

2. Interpretation of graphs with independent curves

Consider figure 2 which shows the time that it takes for three cars to travel a certain distance along a road. The three curves are independent. I asked the pupils questions which required them to interpret the information contained in the graph by comparing the three curves.

I gave written tests to 120 14-15 year old pupils. Protocols of some of these pupils answers were also collected. Analysis of the answers that pupils gave revealed the frequent occurrence of two kinds of errors: concepts related to gradient, and visual distraction, as described below.

(i) Concepts related to gradient

The following extracts from my data illustrate the kinds of errors that the pupils made. After each example I have suggested reasons why the errors occurred. The questions below are from a series of questions which the pupils were asked.

Question: Which car is going faster after:

- (i) 4 seconds
- (ii) 6 seconds

Answers: (i)

	Black	Blue	Red	Can't tell
Freq.	7	4	92	8

Answers: (ii)

	Black	Blue	Red	Can't tell
Freq.	29	10	67	4

Explanation: Some pupils seem to associate fastest with highest. These pupils, therefore, look for a high value. They do not look for the line with the steepest gradient or even for an interval. It is interesting

that fewer pupils were misled by the red car's line in the second question. This is probably because all the lines are closer to the top after 6 seconds than after 4 seconds. In fact, the line representing the black car is highest at  $6\frac{1}{2}$  seconds.

Question: Does black overtake blue, or does blue overtake black?

Some  
Pupils

Responses: K.C. Black overtakes blue. Because the black goes further up the page than the blue.  
G.H. Black overtakes blue. The end of the line is further up the road.  
R.D. Black overtakes blue. Because it is going higher at the end.

Explanation: K.C. associates overtaking with being furthest. G.H. makes the same association as K.C. This pupil (G.H.) is also interpreting the graph very visually; the line is seen as a road. (See the next section on visual distractors for further explanation.) R.D.'s association of furthest and highest is quite obvious.

These results, plus those cited by Janvier (1978) from a similar study with younger pupils (mostly aged 11-12 years) indicate that many pupils are not able to interpret changes in intervals. Pupils who do not have these basic skills will not benefit from using computer simulations which display data in graphs.

(ii) Visual distractors

The term "visual distractor" was used by Kerslake (1977) to describe some graphs which seem to set up strong conflicts for students with poorly developed interpretation skills. These graphs are interpreted in terms of 'concrete' forms which can be related to the situation depicted in the graph. Time and distance graphs, for example, are interpreted by many pupils as if they are a hill or a map.

In the following examples pupils interpreted figure 2 (Three cars travel along a road) as though it was a road (J.S., D.V.) or as though the lines represented cars (C.H., N.B.)

Question: What happens to the red car? (Does it speed up, slow down or what?)

Pupils

Answers: J.S. It turns off to the right.  
C.H. It crashes.

Question: Does Black overtake blue, or does blue overtake black? How can you tell?

Pupils

Answers: D.V. Black overtakes blue. You can tell because the black car went a different route.  
N.B. Black overtakes blue. Because they cross at the same time and then black moves ahead.

Explanation: The answer given by N.B. is a hybrid between interpreting the graph with relation to its content and interpreting it symbolically. Many levels of hybrid answer have been observed. Pupils appear to seek explanations in terms of concrete objects when the interpretation task becomes too abstract and demanding for them.

The message that these results convey is the importance for pupils, of being able to relate the abstract lines of the graph to a familiar situation.

3. Strategies for interpreting graphs with multiple interdependent curves

A pilot experiment was performed in which six subjects were given the graph in figure 3. This graph is adapted from a graph in an Ordinary level, General Certificate of Biology text. The task (question) which the pupils were asked was "describe what is happening in the stream". The pupils had learnt about the biological concepts shown (e.g. photosynthesis) in previous lessons.

Analysis of the pupils' transcripts revealed a range of competence. Very weak pupils tended to treat each curve independently, with little reference to the situation. These pupils usually point to particular points on the curve and do not compare and relate levels and gradients along the curve. This is illustrated by a short excerpt from H.B.'s transcript. After encouragement H.B. began to compare and relate the sections of the curve.

H.B. It it (points at curve for oxygen) starts off high

Researcher. That's fine can you look at it a bit more carefully this time and describe in a little more detail what is happening?

H.B. Well, it goes sharply down and then it goes gradually up.

The other pupils were able to describe gradients relationally. Most pupils explained that the changes in the curves had been caused by the sewage. A few pupils attempted to inter-relate the curves in similar ways to the excerpt from M.D.'s transcript.

M.D. After the sewage goes in the amount of oxygen in the water goes down. This then causes the small green plants to go down. No the amount of

small green plants goes down so the amount of oxygen goes down.

None of the pupils, however, were able to adequately relate the graph to the situation. Very few pupils could extrapolate the curves and suggest that the levels of the organisms and substances would return to their original levels.

Cause and effect relationships were not sought by many pupils and hardly any pupils suggested hypotheses to explain the events shown in the graphs.

The message from this work is that pupils have a great deal of difficulty interpreting these kinds of graphs. They do not inter-relate the curves. There is no reason to think that they would perform differently with computer generated graphs.

4. Conclusions: some guidelines

The results of this work indicate the need for a set of guidelines for authors of CAL materials. Authors should not assume that graphs generated by microcomputers are easier to interpret than graphs on paper. In particular CAL designers should:

1. try to ensure that artifacts are not included which encourage pupils to mis-interpret the graph as a concrete form.
2. not make assumptions about dynamic delivery. Work by Avons et al (Report No.1) does not support the belief that dynamic displays encourage pupils to develop intuitive concepts about gradients.
3. avoid using symbols which will mislead pupils (e.g. one pupil thought that the arrow in figure 3 meant that the sewage was rising).
4. carefully consider the importance of scales. Pupils who are weak at interpreting trends in data tend to read absolute values. This seems to cause an overload of information and impedes interpretation. In any case a V.D.U. display is not suitable for making accurate readings.

REFERENCES

Avons, S.E., Beveridge M.C., Hickman A.T., and Hitch G.J.

Report No.1. Microprocessors in Education Research Project. University of Manchester. "Teaching journey graphs with microcomputer animation: effects of spatial correspondence and degree of interaction. An experimental study".

Kerslake D. (1977) M. Phil. Thesis Chelsea College, University of London. "The Concept of Graphs in Secondary School Pupils aged 12-14 year".

Janvier, C. (1978) Phd. Thesis. University of Nottingham.

"Interpretation of complex cartesian graphs representing situations - studies and teaching experiments".

Tranter, J.A., and Leveridge, M.E.

Chapter 4 Pond Ecology in Computers in the biology curriculum edited by Leveridge, M.E. 1978. Edward Arnold (Publishers) Ltd.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Tim O'Shea for his help and also Marc Eisenstadt and Ann Jones for help and advice on drafts of this paper.

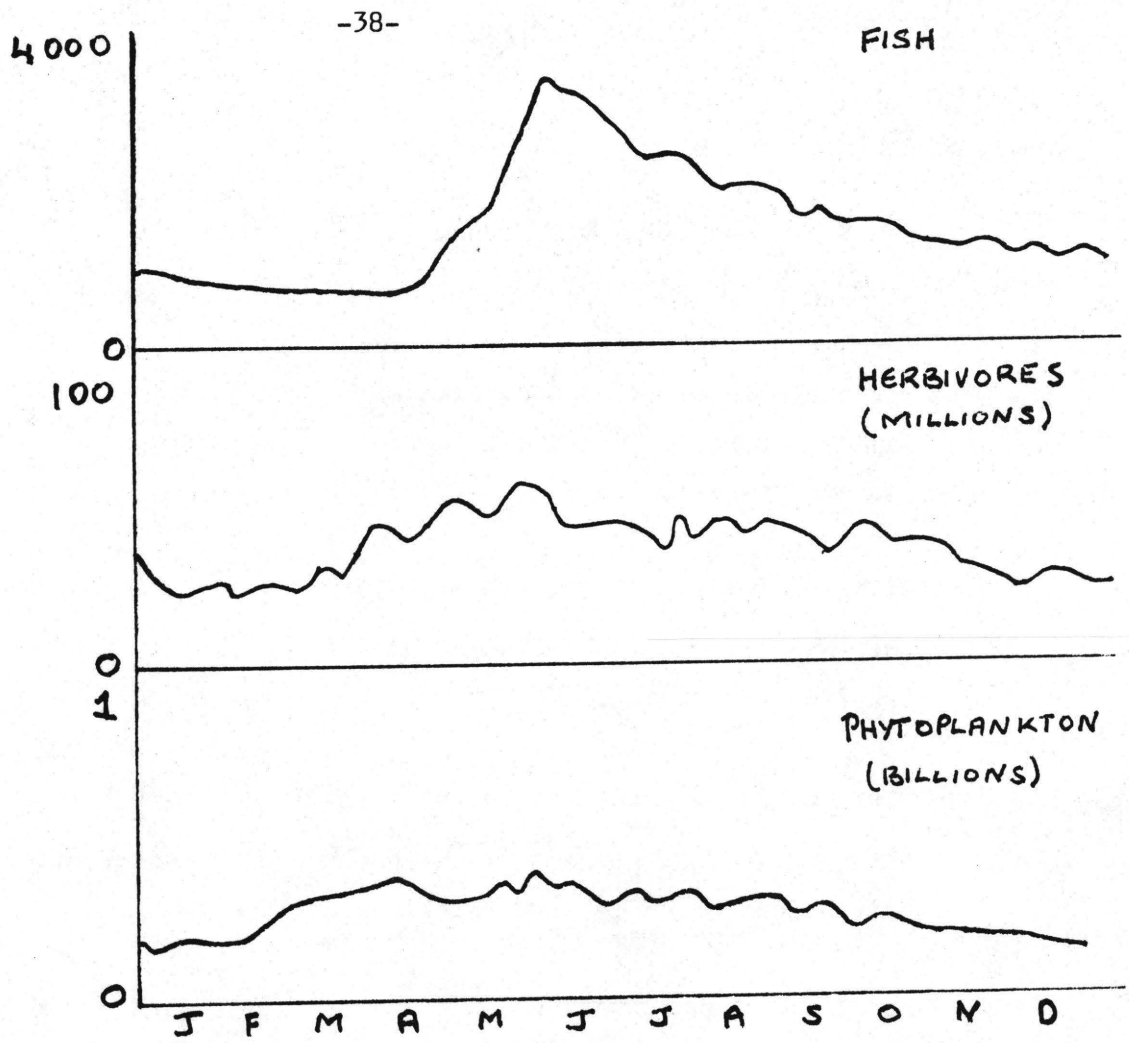


Figure 1. Graph produced by the computer simulation  
POND (Leveridge 1978)



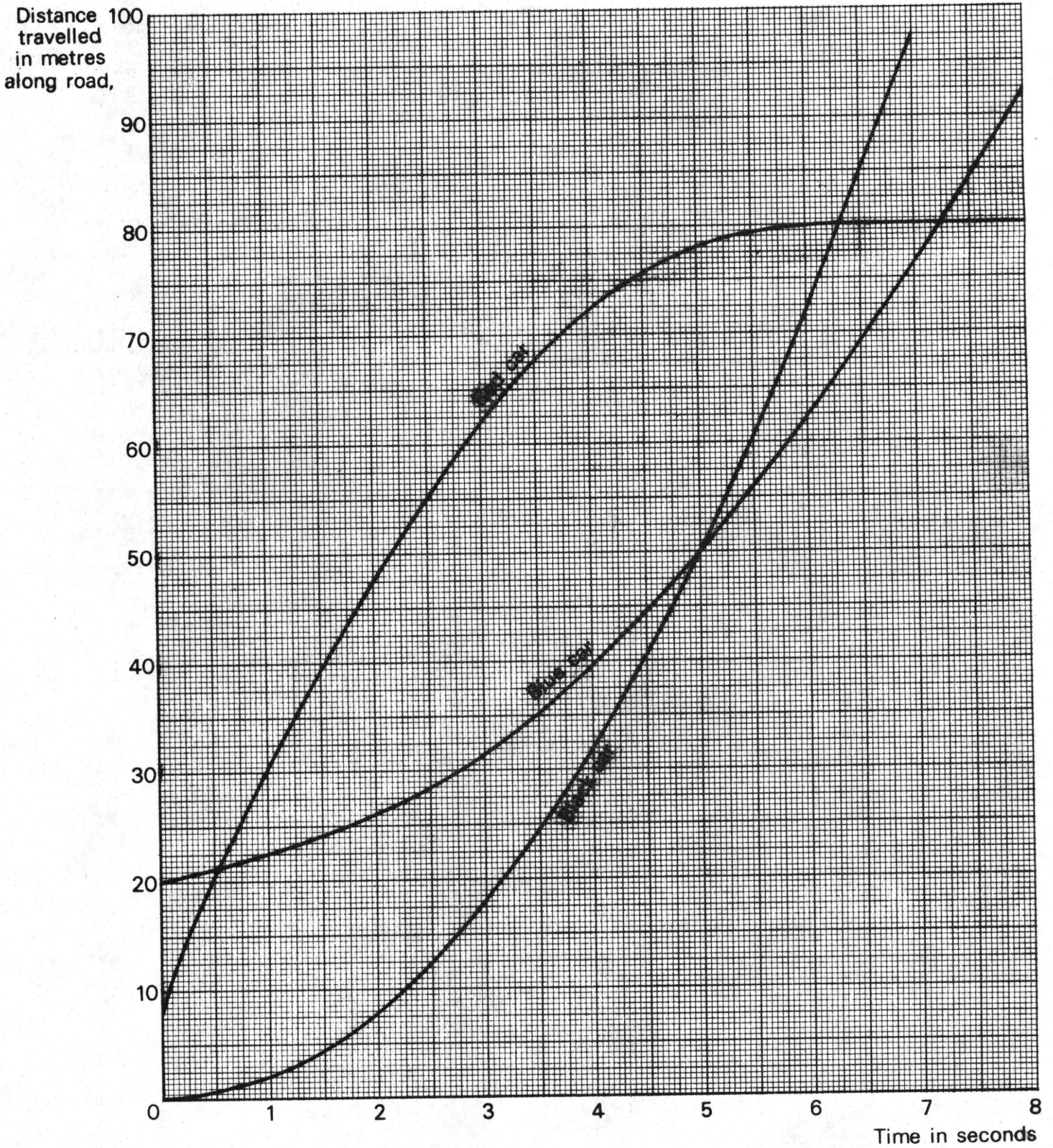


Figure 2 Graph of three cars travelling along a road

(Swan M. The Language of Graphs. Shell Centre for Mathematics Education, Nottingham.)

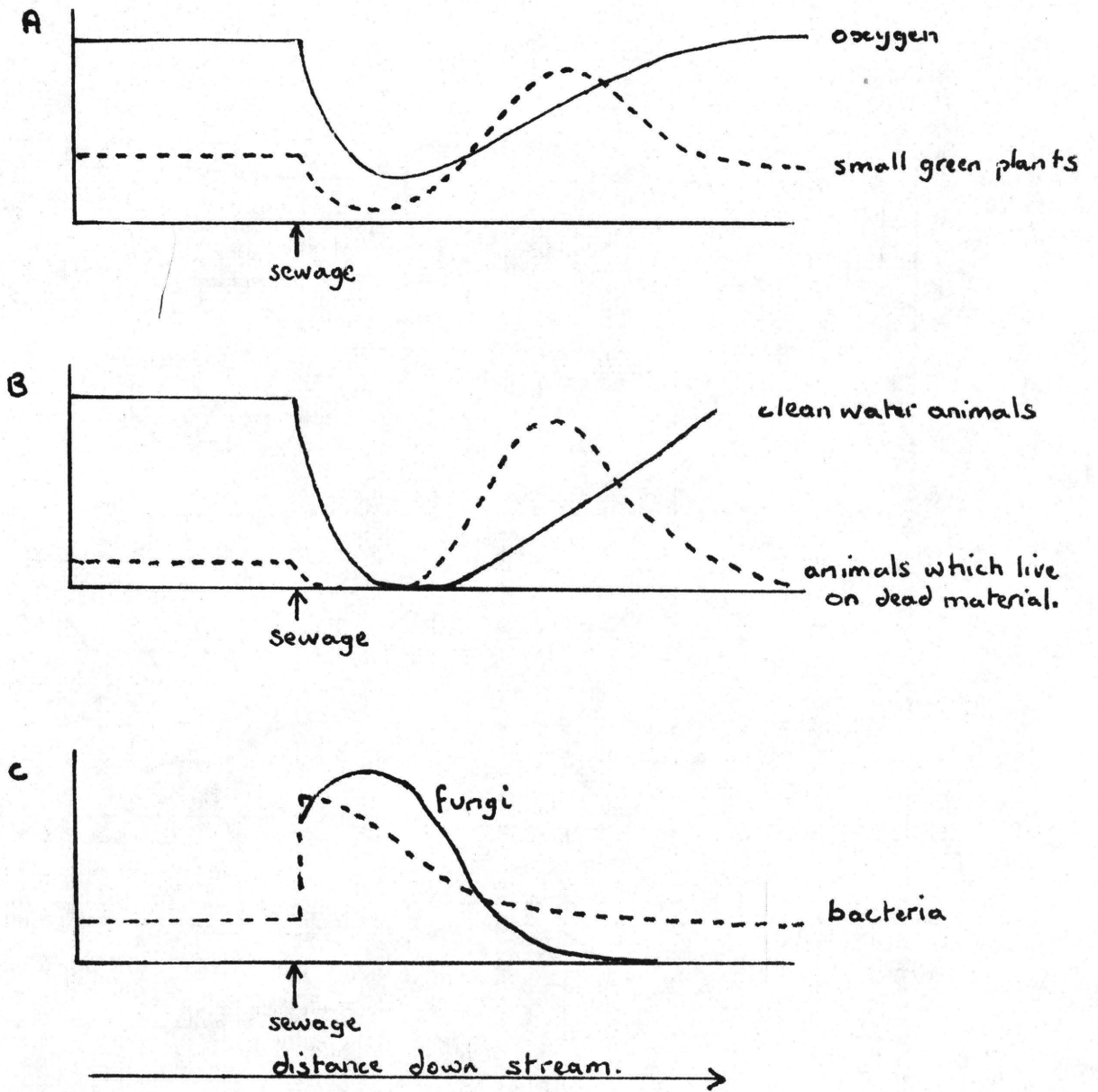


Figure 3. Pollution of a stream by sewage

Steve Payne

The Perception of Grammars: Higher-Order Rules

S. J. Payne

MRC/SSRC Social & Applied Psychology Unit  
Department of Psychology  
The University  
Sheffield S10 2TN, U.K.

1. The Problem

A serious problem in the design of languages for casual users, occasional users, novices, etc. is to make the language easy to learn. The unthinking answer is to construct a small language, as though a small language would necessarily be easier to learn than a big one. The goal of this paper is to explore more satisfactory answers based on psychological principles of organisation and structure, and to relate these answers to the intuitions of computer experts and the findings of software psychologists.

What kind of language? Almost any artificial one. Examples would include: the command languages for word-processors, data base retrieval systems, computer-aided design systems, and other end-user packages; job control languages for large computer systems, and operating system commands for small ones; miniature artificial languages created for experimental purposes; and conventional programming languages. We need not even limit the conception of 'language' to the written mode, as is usual in computing. The action-based languages of calculators and certain graphics systems (Reisner, 1981) can be considered in the same way.

The argument is as follows. Novices learning an artificial language look for guiding principles. They do this because it is more efficient to discover a systematic structure in the language than to learn an entirely arbitrary collection of rules. Various guiding principles have been proposed; the one we shall consider is the principle that the rules of the grammar must have a family resemblance, so that missing rules can be reconstructed by extrapolation from other rules. That means

Steve Payne

that the rules of the grammar are highly organised, which in turn means that they can be derived from higher-order rules, in just the same way that sentences can be derived from ordinary rules of grammar. Thus a new psychological principle is being proposed:

Not only do we learn a language by systematising it into rules, but we also systematise those rules into higher-order rules.

It will be argued that this model of learning artificial languages fits well with the intuitions of experienced programmers, who use such words as 'harmonious' to describe a syntax, and that it also can be regarded as a special case of familiar principles in the psychology of memory and learning.

## 2. 'Guiding principles' in command language design

Research on the learnability of command languages has uncovered a number of 'guiding principles'. Barnard et al. (1981) have shown that novices learning a command system remembered the order of arguments best when there was a consistent order across all command words (guiding principle 1); when there was no consistent order, the next best performance was achieved when the order relied on the pattern of English, in which the direct object usually precedes the indirect object (guiding principle 2).

Ledgard et al.(1980) modified a commercial text editor to make its commands interpretable as short English sentences (guiding principle 2 again). Instead of RS:/TOOTH/,/TRUTH/ the subjects wrote R "TOOTH" W "TRUTH", meaning Replace ... With ..., and performance improved dramatically. Although the "short English sentence" interpretation has some face validity and some support from a study by Wright and Reid (1973), using a rather different paradigm, it has also been argued that the effect was due to making it clearer which symbols were literals and which were delimiters. Potential confusions of that sort, described as character ambiguity by Thimbleby (1981), should be avoided, to simplify comprehension (guiding principle 3).

A different approach has been offered by the work of Mayer (see Mayer, 1979, for an overview) who has shown the efficacy of using concrete images to help learners understand Basic and thus learn it faster (guiding principle 4). Du Boulay

congruence  
hierarchy (kitchen  
chair)

Steve Payne

et al. (1981) extend this argument.

Last in this anthology, Carroll (1980) showed that the internal structure of the rule system exerted a powerful effect, and that rules should be hierarchical and 'congruent' - which is to say that the language should contain familiar bundles; ADVANCE/RETREAT would be a congruent pair for controlling a robot, but GO/BACK (with the same meaning as advance/retreat) would be non-congruent (guiding principle 5).

While more examples could no doubt be found the pattern is clear. Learners need organisation. They can obtain it from familiar concrete prototypes, or from familiar rule systems such as English, or from a perceptible organisational principle within the rule-system. We shall explore the last possibility further.

### 3. Organisation and grammar learning

An effective demonstration of the role of organisation in learning an artificial language was given by Green (1979), who compared dialects of a nonsense language, 'Jabberish', in which the proportions of 'marker elements' were varied. All dialects had the same fundamental structure, with a simple phrase structure, but in some dialects each word was preceded by a marker word peculiar to that class, as though in English all animate nouns were signalled by a preceding word 'animal'. In other dialects of Jabberish, phrases were given their own unique markers, as though in English all noun phrases were signalled by the word 'the'. Finally there were conditions with no markers and with all possible markers. By varying which words and which phrases were marked, a number of delicate predictions could be derived and tested, given the basic assumption that the markers were an effective aid to perceiving the structure of the grammar. The experiment showed that dialects were in general better learnt when markers were present, and more particularly those aspects of each dialect that were signalled by markers were better learnt. Further evidence comes from Reisner (1981), who used an interactive graphics system as a testbed, putting a plausible case for regarding actions at a terminal as being subject to rules of grammar in the same sense as programming languages. One system, called ROBART 1, required fewer control actions of the user at the price of a more complex grammar than the competing system, ROBART 2. In several places Reisner identifies 'structural inconsistencies' in ROBART 1, places where different

Steve Payne

sequences of actions are required depending on what is being drawn when it would be reasonable to presume that identical sequences would be used. Thus in ROBART 1 at least two rules have to be learnt to handle these cases, whereas in ROBART 2 a single general rule suffices. In ROBART 1 users tended to apply the wrong rule, and Reisner compares the problem with the child who has learnt that "verb + -ed makes past tense" and then insists on saying "yesterday I goed". In general users made more mistakes at the points identified by Reisner.

Since ROBART 1 demanded fewer control actions, it seemed to Reisner that rather than minimising the number of terminal symbols in a language - i.e. the lexicon size - one should instead minimise the number of grammatical rules. More exactly, she writes "the number of (forms of) rules", because the key point is to achieve consistency between rules.

The notion of consistency between rules has many attractive features. It ties in with the concepts which computing people use when discussing languages: Basic has a family resemblance with Fortran (though virtually none of the statements are exactly equivalent); the syntax of Pascal is 'harmonious', that of Fortran is not; and so on. Particularly important is their concept of 'orthogonality' in syntax. If one kind of object, say an identifier of type real, is for most purposes similar to another kind of object, such as an identifier of type boolean, then it should be possible and meaningful to use either of them in all logically acceptable contexts. Not so in Pascal, where for example one can read in data values for reals but not for booleans. The syntax is in this respect non-orthogonal.

How would one characterise orthogonality and harmony in syntax? Consider the following diagram:

+++	0	**
OO	***	+
*	?	OOO

Steve Payne

What goes in the cell with a question mark? If it turns out that anything except two plus signs goes there we would feel affronted. On the same basis we feel that the pattern has been broken when we discover that Pascal can read reals but not booleans.

A similar type of organisation within rule systems can be seen in family resemblances. Many rules of grammar in programming languages have the same structure but combine different elements; for instance the notion of sequence turns up repeatedly in the Algol/Pascal family. A program is a sequence of statements separated by semicolons, a declaration is a sequence of identifiers separated by commas, an array declaration requires a sequence of range expressions separated by commas, and an expression is a sequence of operators separated by operands. The notion of sequence is nowhere mentioned in the official grammar, but clearly it is part of the perceived grammar.

4. Higher-order rules can be formalised

A convenient formalism has been invented to describe these family resemblances, the van Wijngaarden two-level grammar (van Wijngaarden, 1966). The language is rather horrifying, as we have to distinguish between proto-notions and meta-notions and between production rules, hyper-rules and meta-rules, but the idea is simple enough. Suppose we have three ordinary production rules,

declaration sequence: declaration / declaration sequence + declaration  
statement sequence: statement / statement sequence + statement  
letter sequence: letter / letter sequence + letter

W  
+ -> >

Because of their family resemblance all those rules can be replaced by a single hyper-rules:

SEQ-ITEM sequence: SEQ-ITEM / SEQ-ITEM sequence + SEQ-ITEM

The term SEQ-ITEM is a meta-notion defined by a meta-rule:

SEQ-ITEM :: declaration / statement / letter

To derive our original three production rules from the hyper-rule, we take the hyper-rule and eliminate the meta-notion SEQ-ITEM from it in all possible ways. First, we find that the meta-rule defining SEQ-ITEM allows it to be replaced by the proto-notion 'declaration'; substituting that for SEQ-ITEM gives us the ordinary

*Ingeniösste Konstruktion  
hin denn!*

Steve Payne

production rule:

declaration sequence: declaration / declaration sequence + declaration

By substituting the proto-notions 'letter' and 'statement' we can recover the original rules. To make this system work, it is ordained that inside hyper-rules all occurrences of one meta-notion (e.g. SEQ-ITEM) are replaced in the same way; whereas the ordinary production rules created from the hyper-rules work in the ordinary way, like Backus-Naur form. (The formalism is well described by Pagan, 1981. To simplify exposition I have not used the convention that 'x' is written as 'letter x symbol', etc.)

While much of the theoretical interest of two-level grammars lies in their ability to express semantics as well as syntax, their interest in the present context lies in their closer resemblance to the grammar-in-the-head. It is unlikely that the grammar-in-the-head corresponds exactly with the two-level grammar, and for that reason the loose term 'higher-order rule' seems preferable to the tightly defined term 'hyper-rule'. We can still feel confident that higher-order rules can be defined more tightly if required. It is also useful to observe that only two levels are required; there is no need for the system to become indefinitely recursive.

5. Organisation and dis-organisation: some examples

Existing command languages for text-editors and word-processors exhibit a variety of organising principles, especially for the control of cursor movements. A good one is WordStar's control diamond:

		E	
A	S		D F
		X	

Control-E sends the cursor up, control-X sends it down; control-S is left by a character, control-A is left by a word; control-D and control-F go right. This happily marries a strong organising principle with the use of perceptual coding, rather than symbolic coding. Notice how "move a long way left" is coded onto a key which is placed a long way left; very ingenious.

Less successful are the symbolic mnemonics used in many other word-processors, such as SpellBinder, Mince (child of EMACS), and Magic Wand.



Steve Payne

Partly that is because one so quickly exhausts the obvious mnemonics: after using F for Forward and B for Backward, what next? Mince uses N for next and P for Previous, V for View-next-page, etc. (In - one presumes - despair, the creators of WordStar used J for Help, adding "That was a Joke. There is no J in Help." It works!)

What we are looking for, however, is relations between rules, not the difficulties of symbolic mnemonics limited to one letter. Mince is interesting because it combines two different organising principles, which seems to make it rather hard to learn. One organising principle is to use the control key for 'small' (e.g. character) and the escape key for 'large' (e.g. word): control-F goes forward 1 character, escape-F goes forward 1 word. But the other organising principle is to use control and escape for forwards and backwards: control-V views next page, escape-V views previous page. This command conflicts with the previous one in two ways, since 'forwards' and 'backwards' are coded differently and 'control' and 'escape' have different meanings.

Most perverse of all, unless the author has simply failed to grasp the principle, is the TXED system, where the only clear thing is that the designers could have used some help, either in choosing a principle or else in explaining the principle they did choose:

	Char	Word	Line	Page	Buffer
Fwd	D	W	N	P	Z
Back	V	U	G	Q	B

These choices have no apparent system of any sort. An interesting two-level rule has also been included: when in command mode the symbol U moves you FORWARD a word!

A number of examples could also be given from the design of conventional programming languages. For instance, in Algol 60, the standard conditional forms were if A then S1 and if A then S2 else S3. The S1 place-holder could be replaced by most kinds of statement, including for-statements, and so could the S3. Does it follow that S2 can also be? Certainly not; it can be replaced by almost any statement, but if a for-statement is used, it must be enclosed by begin and end. While there are valid technical reasons for this restriction (it defends the grammar against ambiguous dangling else-clauses) it is one which prevents the application of the extremely simple and to-be-expected rule that S1, S2 and S3 are exactly

Steve Payne

equivalent slots.

#### 6. Higher-order rules as a special case

There is a strong psychological literature on the learning and recall of word lists and paired associate lists. These paradigms have been used for many purposes, among which we find that the influence of intra-list structure has been extensively studied. Of course, there are many other aspects of organisation in memory and many other paradigms for its investigation (Puff, 1979, *passim*) but the paired-associate list seems remarkably similar to the list of commands paired with their effects in a command language.

The ubiquity and pervasiveness of organisational principles can be seen in Tulving and Donaldson's important book (1972). The standard result is that a list of items which can be grouped into categories (flowers, animals, etc) is better recalled than one which is not categorisable. This effect depends on the number of categories and the number of instances (Weist, 1970). Not surprisingly, it is also clear that subjects impose their own coding schemes when necessary, and these can also improve recall (Tulving, 1962). One particularly interesting result is the "some-or-none" phenomenon: either several items from a given category are recalled or else none at all (Cohen, 1966). That clearly bears on our interest in recreating forgotten or unknown commands from the ones that can be recalled.

In an entirely different experimental paradigm, it has also been shown that transfer from one miniature artificial language to another is easier when the languages bear a family resemblance, even when the subjects were 'unaware' of the resemblance at the verbal level (Reber, 1967). In another valuable finding Reber et al.(1980) showed that the structure of a simple rule-system could be disguised by one type of display and revealed by another, to the extent that subjects could categorise candidate sentences as grammatical or not, although possibly not having a verbal formulation of the rule. Moreover, telling subjects to look for a rule did not necessarily help, as the conscious search tended to lead them astray; the best results came when the existence of a rule was revealed after presentation of examples, rather than before.

Although the effects of organisation on memory have been actively studied for some time, their theoretical interpretation is still a matter for discussion (Voss,

Mince!

-49-

An experiment on organisation  
and dis-organisation.

• can word-total here  
• 4 tasks

Steve Payne

1979). Nevertheless, for immediate practical purposes it is safe to assert that organisational effects in the perception and learning of rule systems are not a novel psychological phenomenon, but a new manifestation of familiar principles common to all aspects of memory research.

### 7. Towards empirical tests

There are no studies known to the author treating the issue directly. An experimental programme is being initiated. The effects of organisation on command languages will be studied by comparing rule systems with a coherent principle, with conflicting principles (as in the Mince example above), and with no organising principle; in the first study, the speed of learning the command sets will be compared, and in subsequent studies their usability in practical contexts will be studied.

A second set of studies will take up similar issues in the design of programming languages. Because of their much higher degree of statement complexity, these will provide an excellent counterfoil to command language studies. It is hoped to demonstrate that when a language reducible to a small number of higher-order rules is easier to learn than a less reducible language, even if the vocabulary or the average sentence-length is greater. Finally, in view of the result obtained by Reber et al. (1980) - not to mention anecdotal comments by people who failed to notice organisational principles in WordStar and elsewhere - it will be necessary to look for effective ways to display higher-order rules. Should we teach generalisations about rules before teaching rules, or afterwards? Or should we not teach them at all, and let the subjects infer them?

### 8. Conclusion

It would be hard to deny that language systems will be more easily learnt when their structure is easy to perceive and regenerate; but words like 'structure' and 'organisation' have proved slippery and elusive in the past. By relating them to the well-defined van Wijngaarden two-level grammar it seems that we can fare better.

At present it seems that a good way to make a language system easy to learn

Steve Payne

is to design the grammar so that it can easily be reduced to a small number of higher-order rules. We hope to report empirical tests on this matter in the near future, dealing with both command languages and programming languages. There are however a few other matters that need to be considered. In particular, some thought must be given to the problem of making the structure visible to the learner. Moreover, it must not be forgotten that several other 'guiding principles' were mentioned above, and these too should be brought into service if we wish to design a very easy-to-learn language.

### References

- Barnard, P. J., Hammond, N. V., Morton, J. and Long, J. (1981) Consistency and compatibility in command languages. International Journal of Man-Machine Studies, 15, 87-134.
- du Boulay, B., O'Shea, T. and Monk, J. (1981) The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies, 14, 237-250.
- Carroll, J. M. (1980) Learning, using and designing command paradigms. Report no. RC 8141, IBM Watson Research Center, N.Y.
- Cohen, B. H. (1966) Some or none characteristics of coding. Journal of Verbal Learning and Verbal Behavior, 5, 182-187.
- Green, T. R. G. (1979) The necessity of syntax markers: two experiments with artificial languages. Journal of Verbal Learning and Verbal Behavior, 18, 481-496.
- Ledgard, H., Whiteside, J. A., Singer, A. and Seymour, W. (1980) The natural language of interactive systems. Communications of the ACM, 23, 556-563.
- Mayer, R.E. (1979) A psychology of learning BASIC. Communications of the ACM, 22, 589-593.
- Pagan, F. G. (1981) Formal Specification of Programming Languages: A Panamoric Primer. N.J.: Prentice-Hall.

Steve Payne

- Puff, C. R. (ed). (1979) Memory Organization and Structure. New York: Academic Press.
- Reber, A. S. (1967) Implicit learning of artificial grammars. Journal of Verbal Learning and Verbal Behavior, 6, 855-863.
- Reber, A. S., Kassin, S. M., Lewis, S. and Cantor, G. (1980) On the relationship between implicit and explicit modes in the learning of a complex rule structure. Journal of Experimental Psychology: Human Learning and Memory, 6, 492-502.
- Reisner, P. (1981) Formal grammar and human factors design of an interactive graphics system. IEEE Transactions on Software Engineering, SE-5, 229-240.
- Tulving, E. (1962) Subjective organization in free recall of "unrelated" words. Psychological Review, 69, 344-354.
- Tulving, E. and Donaldson, W. (eds) (1972) Organization of Memory. New York: Academic Press.
- van Wijngaarden, A. (1966) Recursive definition of syntax and semantics. In T. B. Steel (ed.), Formal Description Languages for Computer Programming. Amsterdam: North-Holland.
- Voss, J. F. (1979) Organization, structure, and memory: three perspectives. In Puff (1979).
- Weist, R. M. (1970) Optimal versus non-optimal conditions for retrieval. Journal of Verbal Learning and Verbal Behavior, 9, 311-316.
- Wright, P. and Reid, F. (1973) Written information: some alternatives to prose for expressing the outcome of complex contingencies. Journal of Applied Psychology, 57, 160-166.



Developing an Instrument for the Analysis of Cognitive  
Activities of Workers at Dialogue Systems

Erhard Nullmeier and Karl-Heinz Roediger  
Technical University of Berlin

Abstract: The following article reports about some research work done at the Technical University of Berlin in order to obtain design criteria for man-machine dialogues. The starting point of the investigation is the examination of the working process. With the results of industrial and cognitive psychology we intend to create an effective instrument for measuring the cognitive activities of those users at display terminals.

1 Introduction

Many working activities are supported by EDP through introducing computers in the field of administration; as a result however working people become dependent on EDP through changes of production scheduling. This leads to psychological stress through Taylorized activities and resultant decrease in the perceived value of the work, through the pace dictates of the machine, through regulation of working activities by the machine, through loss of communication among the workers etc. These problems very often are understood as temporary. We believe an analysis of these psychological changes is needed, and such an analysis has not as yet been conducted.

Since June of this year an interdisciplinary research project is supported at the Technical University of Berlin, which shall attend to the investigation of these questions. The aim of this research project is to develop fundamentals for the design of dialogue interfaces in computer assisted systems under criteria of industrial and cognitive psychology. Computer scientists and psychologists work together in this project, for that a term of four years is planned. As well as two newly created full-time positions are held by com-

puter scientists, who also work in the borderlands of psychology for some years, five computer scientists and five psychologists contribute to this project with portions of their working time.

As the project is in statu nascendi at the time this paper must be written, we cannot present results. On the contrary we want to report about our aims and our methods as well as about first problems as they have cristallized out of the preliminary studies for this project, which took almost a year.

In the following we first of all intend to characterize the problem as accurately as possible and thereby emphasize on those dialogue places of work, tasks, and cognitive activities, which shall be the subject of our investigations, and which in our opinion are amenable to software design. In the subsequent chapter we go into the particulars of the state of the art. With the walking along critique of the existent positions, we want to point at their limitations and their weakness, which we hope to avoid. After that we outline our methodical positions; also here we can offer more problems than answers. Finally, as a prospect, we will refer to the possible and desirable results of this project for computer science, for psychology, and - last not least - for the users of computer assisted systems.

We are aware, that this project cannot solve the problems of computer assisted places of work; for that the power of social pressure groups is needed as well as competence. Nevertheless we believe, that there is a latitude for designing technology, which should be used to achieve improvements in a limited domain. We hope to carry out not only a contribution to the enhancement of acceptance, but at the same time we make up our minds to point out the limits of computer application.

## 2 Starting Point

In this interdisciplinary research project we intend to investigate dialogue places of work in the field of administration of public services as well as free economy. We understand a dialogue work place to be not only a place for storage and retrieval of da-



ta but also for other cognitive activities belonging to the working process as for example, the decision about the allocation of a credit in a bank or the adjustment of damages in an insurance. In this early stage of our project we leave undecided whether the working process which we will investigate runs fully or only partially automated. As we proceed from the hypothesis, that one part of the dissatisfaction of workers at dialogue places of work results out of the shift of decision competence from man to machine, we want to become acquainted with both types of work places. Among other things we hope to obtain out of this, significant statements about designing technology regarding cognitive requirements, which the working process should assign to the worker.

Following another thesis, that in the main cognitive activities are designable through software, we confine ourselves to exactly those aspects of the work at dialogue places of work as the main part of our investigations. Further components of the working activity, such as affective or sensumotorical ones will be considered at least in the explorative interviews of the initial period of this project in order to obtain statements about exactly that part of the working activity one can design by software. The essential of that is to determine how and to where interpersonal communication at work places has been switched over. But it is not our aim to reconstruct lost communication between colleagues by displays and software design.

On the contrary our aim is to analyze cognitive activities such as the collecting, processing, and evaluation of information, as well as computations and decisions in order to obtain criteria, which parts of these activities can be delegated to the computer without interfering into autonomy and competence of the worker to such an extent, that he becomes a part of the omnipotent machine respectively feels himself like that.

### 3 State of the Art

Two recently published papers, the article of MORAN on "An Applied Psychology of the User" (/6/) and the monograph of SHNEIDERMAN on

"Software Psychology" (/9/) can be regarded as typical investigations of human behavior when working with dialogue systems.

The authors report on psychological investigations about isolated aspects of working at display terminals. They often try to optimize the man-machine system, whereby they look upon humans as black boxes for reasons of simplicity. They examine their input and output disregarding the nature of information processing in the human brain. In attempting to explain human reasoning intricate cognitive processes are simulated in a simple laboratory environment.

MORAN wants "more than just explaining user behavior ... to predict and to control it" (/6/, p. 3). He focusses his attention exclusively on the pure cognitive aspects of the individual user. According to his aims he prefers calculational models of user behavior.

Floyd, Keil, and Nullmeier state that from this point of view "users (= people) are reduced to (machine-like) error-prone components of the human-computer system" (/4/, p. 491). MORANs approach "implies that only those dimensions of user behavior which are measurable can be considered. Predicting and controlling user behavior on the basis of the models thus obtained will contribute to increasing the performance of the system at the level of individual transactions, but actual user needs will not be taken into account. Even if system performance is the overriding goal, this approach will produce misleading results, since the overall system performance will only be satisfactory if the system does not force the user either to decompose his work into awkward steps or to work against or around the system in order to maintain a reasonable work style" (/4/, p. 491).

These critical comments can be supported through the dissatisfaction of psychologists like NORMAN (/8/) with research on cognitive psychology. He finds fault with the narrow limits of investigating only pure cognitive systems, and noticed that "the results (of this approach, the authors) have been considerable progress on some front, but sterility overall, for the organism we are analyzing is conceived as pure intellect, communicating with one another in logical dialogue, perceiving, remembering, thinking where appropriate, reasoning its way through the well-formed problems that are encoun-

tered in the day. Alas, that description does not fit actual behavior." (/8/,p. 4).

NEISSER (/7/) emphasizes that the human intelligence which can be observed daily when humans work or simply live, was hardly ever investigated by psychologists. He states that "anyone who wishes to study such problems must start from scratch ... It is an example of a principle that is nearly as valid in 1978 as it was in 1878: If X is an interesting or socially significant aspect of memory, then psychologists have hardly ever studied X." (/7/, p. 4).

Summarizing we mean that the previous results of cognitive psychology are of less value to our project; we hope that a pressure from applicants and clients of psychological research will lead to more investigations on practical aspects of cognition.

Despite these critical comments we intend to "study applied psychology in order to find out how to make computer-based systems more adequate as tools for their human users. Taking this approach, one would focus on such questions as 'How should computer-based systems be embedded into the working context of their users ?' 'What kind of computer system models are appropriate for the users ?' 'How can we contribute to making systems more transparent to users ?' 'What does transparency mean, psychologically ?' 'How can individual transactions and streams of transactions be best arranged to correspond to meaningful work steps ?' ...

Taking this approach, we also will start by considering individual transactions, but we will pose different questions and interpret the answers in a different context." (/4/, p. 491f.).

DZIDA (/2/) reports on some research to support the human working process at dialogue systems by appropriate tools. He compares complex software tools, which are difficult to understand for the user, with simple software tools, which can be combined to more complex tools by the user in a modular way. DZIDA states, that in consideration of the principles of the regulation of human actions the latter kind of tools are more adequate for humans.

Now we want to mention some psychological work that can be a starting point for our project. The human working process can be seen as goal-oriented, rationally, consciously, and mutually dependent on other humans. The human decides what his aims shall be (at least

what shall be partial aims through which he can reach the given goal) and which of the available tools he wants to use. The performance of the task can be seen essentially as rationally. This latter assumption means simplifying the human worker, but this objection can be weakened because the planned analysis of the working process will take into account the given task as well as the real individual working process. Therefore the analysis includes non-rational elements of human behavior, too; e.g. the individual preference for certain sequences of transactions, and the way of using external storage space for data, which will be further processed by the individual or his colleagues.

Mutually dependent on other humans, means that the individual produces and finally consumes, and that his working and living conditions are influenced by other humans. These assumptions, which are essentially different to those of classical industrial psychology, are explained in detail by VOLPERT (/10/). On the basis of these ideas and some formalism which were taken from computer science, mainly HACKER (/5/) developed a theory that explained the performance of actions focussing on the psychological regulation of the sequencing of actions. The main application area of this theory are industrial working processes, e.g. the control and supervision of partly automated manufacturing processes.

Clearly, there remain some doubts as to what extent this theory can help us solve our problems. What we mean is that the basic assumptions of the nature of the human working process are a suitable starting point to raise other questions. These assumptions are appropriate when they concern the aspects of working with dialogue systems, which can be influenced by software design. Additionally, the aspects of planning cognitive activities and of task-oriented communication must be integrated into the theory.

Since we are planning to analyze the working process at dialogue systems, the earlier developed instruments on the basis of this theory can give valuable hints. As mentioned by FISCHBACH and NULLMEIER (/3/) the instruments of Baarß et.al., Mickler et.al., and the VILA are of main interest.

In the following section we want to explain our research schedule, that is based on the mentioned theory and especially on critics of this theory.

#### 4 Our Positions

We are planning to develop an instrument for the analysis of cognitive activities. This instrument shall be constructive in the sense of getting design criteria out of the results of the analysis, and we are hopeful of being able to rate design alternatives. We hope to combine various developments in the field of industrial psychology, particularly those of cognitive regulation of working activities, with models and results of cognitive psychology.

First of all we must select appropriate dialogue systems and ensure that we can investigate the users of these systems. In the selected administration area we plan to perform explorative interviews with the users. These interviews should help to prepare the planned detailed analysis of cognitive activities. We emphasize questions concerning the changes of the structure of tasks and cognitive activities that are influenced by EDP, especially by dialogue systems.

We intend

- to find out, what common and what distinct aspects characterize the working activities. Thus we want to define the group of dialogue systems and persons, which should be investigated further; simultaneously we want to estimate, to which similar places of work the results of our analysis can be transferred.
- to find out, what are the real users needs and interests; thus we gain some hints, at least what we must include into our analysis. The interviews should also answer the question, what questions can be answered by the users.
- to get comparable statements concerning the quality of dialogue systems as seen by the users, and demands for a better design of interfaces.

The assessment of different realisations of dialogue systems and therefore of working activities should take into consideration to what extent these activities promote the individual personality. Essential here is that the individual comprehends and appreciates the entire working process, and not merely a compartmentalized portion of it. An other criterion is the influence of the worker

on his working process, particularly in the independent definition of partial aims and the selection of his own tools.

An analysis of the demanded activities to perform a given task, especially of the mentioned aspects and the relations between them, may help to detect incomplete (or partialized according to VOLPERT (/10/)) activities. A demanded activity is called incomplete, if only a small part of the levels of regulation is necessary to perform the activity. As a particularly pronounced example, data input tasks do not make demands of the abilities of humans to organize mentally their own working process. Taking the aspect of communication into account, an activity may also be incomplete, if the communication between humans is restricted to the credit of the man-machine dialogue.

Instruments for the analysis of working activities typically show the weak points of the activities; since an activity which promotes the individual personality in an ideal way cannot be defined conclusively we must start from these weak points (negative demands). DÖRNER (/1/) investigated the problems of humans to handle complex decision situations. He states as a result of this project, that merely the faults that humans made from his viewpoint as the supervisor of the experiment can be observed.

In our project we are planning to investigate to what extent the degrees of freedom to perform the task are restricted or enlarged. Examples for restrictions are humans who are forced to store all their knowledge of the working process into the computer, who are forced by the system to decompose their work into awkward steps, and who are restricted to communicate with other humans.

The degrees of freedom for the human to define his own aims and to plan his working activities on his own responsibility can be analyzed through such parameters as the number of alternatives to reach given goals, the definition of subgoals and subtasks, the sequence of performing subtasks, the possibility to get a feedback concerning his activities, ...

When working with dialogue systems cognitive activity can be characterized by the way in which and the frequency with which software tools are used. In particular we should find out how far the user

can see through these software tools in that he creates more complex tools out of simple elements corresponding to his progress in learning and to his cognitive style.

On account of the use of dialogue systems the cognitive process of the users of these systems can be examined using dialogue protocols and data files which are changed by the user. The possibility of defining his own aims and the cognitive process are mutually dependent; we intend to investigate what consequences have certain restrictions concerning the definition of goals, e.g. according to internal standards to structure a formular, to the variety of cognitive activities.

This method of analyzing "documented cognitive processes" must be completed by an interrogation of the worker. By this combination of objective and subjective elements in the analysis we hope to find out the relevant cognitive activities that lead to the decision process, the task-oriented communication that take place or not, and the reasons of using certain software tools.

## 5 Aims

We are aware, that this project begun by us moves along the small edge between enhancement of acceptance and rejection of new technologies. It will be our function to test the edge especially in order to find out which parts of the working activity are approachable to software design. We hope to become effective in two social domains with the instrument for the analysis of cognitive activities, which we want to develop.

In the field of science we hope to expand the existing psychological theories in such a way, that these new activities become comprehensible and measurable with those theories. For the computer scientists we hope to create verifiable criteria, with which they can begin the design of interfaces in the interest of those persons affected by these places of work.

We hope to utilize our knowledge about the effects of application of these technologies together with the affected persons in the sense, that they can engage themselves in the designing process by

formulating the requirements upon their work places. Already at the beginning we have noticed, that we cannot put to an end all problems of computer assisted work places with this. We also see the risks, which are located in an instrument such as that one we will develop for the analysis of cognitive activities: it can be misused as a contribution to the continuing partition of cognitive activities. At this point we are obliged to call attention to the limits of scientific research and we are obliged to take not longer part in the process of distortion of human working activities by taylorizing those activities and by intensification of the work pace.

Before we come to the point, where we only have to show the limits of computer application, we believe to find that play, where it is possible to design technology in the interest of the persons affected by this.

## 6 References

- /1/ DÖRNER, D.  
Über die Schwierigkeiten menschlichen Umgangs mit Komplexität  
Psychologische Rundschau 32 (1981) No. 3, p. 163-179
- /2/ DZIDA, W.  
Unterstützung menschlichen Arbeitshandelns durch das Datensicht-  
gerät  
in: Ergonomic (Ed.), Das Datensichtgerät als Arbeitsmittel,  
Berlin 1981
- /3/ FISCHBACH, D. and E. NULLMEIER  
Nutzen von Arbeitsanalyseverfahren in Abhängigkeit von der  
theoretischen Konzeption  
to be published in: FRIELING, E. (Ed.), Konzeptionen und Mo-  
delle der Tätigkeitsanalyse, Frankfurt am Main 1982
- /4/ FLOYD, C., R. KEIL, and E. NULLMEIER  
Letter to the editor-in-chief of ACM Computing Surveys  
ACM Computing Surveys 13 (1981) No. 4, p. 491-492
- /5/ HACKER, W.  
Allgemeine Arbeits- und Ingenieurpsychologie  
Berlin 1980
- /6/ MORAN, T.P.  
An Applied Psychology of the User  
ACM Computing Surveys 13 (1981) No. 1, p.1-11



- /7/ NEISSER, U.  
Memory: What are the Important Questions ?  
in: GRUNEBERG, M.M., P.E. MORRIS, and R.N. SYKES (Eds.),  
Practical Aspects of Memory, London 1978, p. 3-24
- /8/ NORMAN, D.A.  
Twelve Issues for Cognitive Science  
Cognitive Science 4 (1980), p. 1-32
- /9/ SHNEIDERMAN, B.  
Software Psychology  
Cambridge, Mass. 1980
- /10/ VOLPERT, W.  
Die Lohnarbeitswissenschaft und die Psychologie der Arbeits-  
tätigkeit  
in: GROSKURTH, P. and W. VOLPERT, Lohnarbeitspsychologie,  
Frankfurt am Main 1975, p. 11-196



I. THE PSYCHOLOGY OF THE COMPUTER USER

B. The programmer



Individual differences and aspects of control flow notations

Gerrit C. van der Veer, Vrije Universiteit, Amsterdam

Jan van de Wolde, Technische Hogeschool Twente, Enschede

1. INTRODUCTION

There has been quite some discussion recently about the urgency of incorporating computer literacy in our school curricula. An argument frequently adduced in support of such a development is the notion that the experiences of computer programming turns people into better problem solvers. Some authors claim that such a transfer is very general in nature, and applies to widely divergent domains of knowledge. A prominent advocate of this view is Seymour Papert (1980). His brainchild, the LOGO-project, springs from the idea that 'doing' is a necessary condition for knowledge acquisition, and that problem solving performance will profit from the procedural approach stressed in computer programming. We must note however that the evidence for this hypothesis, provided by Papert, has not been very convincing up to now. Whether learning LOGO-programming, and elaborating its conditional branch statement, effects conditional reasoning ability in children has been the subject of a study by Seidman (1982). His experiments, designed more powerful than those of Papert, suggest that such effects, if present, are very limited in nature and strength. Less ambiguous positive transfer effects of computer programming have been reported by Soloway & Lockhead (1982). The results of their experiments indicate that mathematical principles are handled more easily when they are to be expressed in program text rather than in plain algebraic terms. The authors warn against danger that such an effect may be undone by the "current push in the direction of more formality in programming; programming languages are in danger of becoming like mathematics".

The fact that notational formats play an important role in understanding mathematical concepts also became clear to us a couple of years ago, albeit in a rather odd way. In an experiment by Van der Veer & Otterangers (1976) simple mathematical algorithms had to be developed in terms of a so-called Pupils Programming Language. It appeared that those students who were apt to avoid math as a study subject, severely handicapped themselves by using meaningless identifiers and by abbreviating basic symbols, whereas math-minded subjects created maximum semantic transparency in the code. By obscuring the code, the first group transformed the originally meaningful problems into a collection of incomprehensible tricks.

It is evident, however, that the importance of computer literacy does not solely depend on transferability of skills. Computing is useful for its own sake. It offers extra opportunities to organise and communicate knowledge, and may therefore augment human intelligence in problem solving and decision making. Moreover, we have to acknowledge the growing social and economic needs for computer literate citizenship in our modern society. Programming as a tool in education is growing more and more popular. In Amsterdam at least 12 primary schools have a computer terminal in one of the classrooms at which a language is available that is especially constructed for children of about 11 years of age. In a growing number of secondary schools informatics is part of the curriculum. At the moment this is only the case on a voluntary base, but it is to be expected informatics will be incorporated in the official curriculum within a few years. In a growing number of university-faculties informatics or at least the use of computers is becoming a normal part of the study.

2. LEARNING A COMPUTER LANGUAGE

When somebody is introduced to computer programming for the first time, there are at least four factors that affect the learning process:

a. learner characteristics

What are the learner's intuitions about computer programming?

How are relevant concepts organized in his mind?

What is his style of representation?

Here we distinguish between three sources of variation:

- abilities; i.e. mental capacities like general intelligence, which are supposed to be relatively stable over time;
- educational background; e.g. previous training in mathematics or logic;

-cognitive style; the way a learner is apt to attack certain problem situations. The adequacy of a style will depend on the situation or the nature of the problem involved. Style as an operational mode is a product of past learning experiences, and may to a certain extent be influenced by directed training.

b. characteristics of the problem solution, i.e. the algorithm to be constructed.

What kind of conceptual organisation is required in order to solve the problem? The problem (solution) may ask for some kind of hierarchical organisation, and some kind of organisation may be suggested by the semantics that are present in the wording of the problemdefinition.

c. features of the programming language

What kinds of conceptual organisation may be expressed in the code? What kinds of organisation are promoted by the syntax? Being psychologists we do not pretend to be exhaustive in our analysis of language features. We will deal one by one with syntactical aspects like flow of control, datastructures or the vocabulary of basic symbols and identifiers.

d. didactics

How do we attain a match between programming constructs and routines of information processing present in the learner's mind? This question concerns the use of modularisation techniques, top down or bottom up programming, structured diagrams and the like.

From these factors two have gained most of our attention up till now, namely learner characteristics and aspects of control flow notation.

### 3. LEARNER CHARACTERISTICS

Our concern with learner characteristics stems from research in our laboratory regarding individual differences in cognitive styles.

#### intelligence

About including intelligence in our set of relevant learner characteristics we will be very short. Intelligence is a very complex concept in which many aspects may be distinguished (e.g. see Guilford & Hoepfner, 1971). We all know that general intelligence, whatever operationalisation is being used, is a strong predictor of very different kinds of problem solving performance. In our experiments we felt content measuring this factor by means of Raven's Progressive Matrices, and next partialling it out from all effects of interest.

#### educational background

One aspect of educational background at least in the Dutch schoolsystems, appears to be of great relevance when a computerlanguage has to be learned. Students that leave secondary school show considerable differences in their apprehension of mathematics. Some students, to be called 'alphas', are only exposed to very simple, introductory courses, and often have deliberately avoided any continuation course since they don't feel any affiliation with the subject matter. Others, so-called 'betas', have received a substantial amount of practice in dealing with formal notations, in symbolic manipulation, and in applying algorithms (not so much in constructing these). Both types of students share the same circumstances in higher education, for instance in faculties like linguistics or social sciences. Their curricula often imply statistics and several kinds of computer use. When we want to examine the feasibility of programming constructs for an introductory computerlanguage, the difference in entry characteristics between alphas and betas will have to be kept in mind.

#### cognitive styles

Pask (1976) has developed some fruitful ideas in the field of learning and teaching strategies. A guiding principle in his work is the idea that educational methods are most efficient when tailored to the individual competence of the student. In view of this notion Pask is particularly referring to partly automatic, intelligent teaching systems. Basic to the strategies are the individual learning styles or dimensions of 'competence' as Pask calls them. These styles reflect modes of organising the acquisition, storage and retrieval of knowledge. Pask designed several devices to measure aspects of these styles, some of which has been translated and elaborated in our laboratory. Most effort has been invested redesigning and standardizing the Smuggler's Test. The plot of this test is a story about a gang producing and trading narcotics. The student

is asked to imagine that he embodied in some international police organisation with the assignment of rolling up this gang. From the way the students organises and reconstructs or reproduces the data, we infer three learning style factors (Van der Veer & Van de Wolde, 1982).

factor I : inclination to learn and to put effort into memorizing. This factor does not reflect an ability. While the instructions do not necessarily suggest that the material should be memorized, some students do so spontaneously and consistently.

factor II : operation learning. This factor concerns the inclination to deduce specific rules and procedural details. It results in the availability of materials that may serve in the construction of procedures, and it is expressed in consistency of references to related details of the knowledge domain.

factor III: comprehension learning. This style reflects the inclination to induce general rules and descriptions, and to record relations between different, or even remote parts of the domain. It is expressed in the tendency to reconstruct lost details by application of general rules and analogies.

#### 4. CONTROL FLOW NOTATION

Our interest in different forms of Control Flow notation was excited by the work of Sime, Green & Guest (1977) in Sheffield concerning conditional branching. In a series of experiments these researchers have compared a number of alternative branching devices as to their 'cognitive ergonomic' properties. It concerned two familiar conditionals which were called JUMP and NEST-BE and a prototypic construct called NEST-INE designed in Sheffield. JUMP is a GOTO-branching statement like the one we know from such languages as FORTRAN and BASIC:

```
L1      IF ILL GOTO L
        DO YOUR DAILY WORK
L1      STAY HOME AND RECOVER
```

NEST-BE (Begin-End) is a control structure like IF ..THEN..ELSE.. in ALGOL-60 supplied with scope markers:

```
IF ILL THEN
BEGIN STAY HOME AND RECOVER
END
ELSE
BEGIN DO YOUR DAILY WORK
END
```

NEST-INE is a related control structure but it provides more redundant information about the conditions that have to be met in order to have some action executed:

```
IF ILL STAY HOME AND RECOVER
IF NOT ILL DO YOUR DAILY WORK
END ILL
```

The feasibility of these structures was tested by comparing several aspects of programming performance when using a special-purpose 'microlanguage' featuring nothing but the construct in question. Summing up the results, the authors conclude that 'by and large these experiments have favoured the NEST-INE dialect (Green, Sime & Fitter, 1981). Nevertheless there was some evidence in their data that raised doubts as to the limits of queuing decisions when parsing NEST structures. Sime et al experimented up to a maximum of 3 levels of embedding and noticed a considerable decline in programming performance with increasing depth.

#### 5. RESEARCH IN OUR LABORATORY

##### design and procedure

Our experiments primarily aimed at examining the interactions between learner

characteristics, language features and some relevant attributes of algorithmic structure. Furthermore we wanted to verify the queuing restrictions supposedly applying to NEST-structures. With regard to language features we decided not to compare all three control-structures, used in Sheffield, but to juxtapose JUMP with the most favourable nesting-alternative, i.e. NEST- INE. The syntax of both languages was defined as follows:  
The NEST-syntax:

```
<program> ::= <action>/IF <condition><program> IF NOT
              <condition><program> END <condition>
<action> ::= <simple action>/<simple action> AND <action>
<simple action> ::= A1/A2/A3/..../A10
<condition> ::= C1/C2/C3/..../C8
```

With the context-dependent restriction thest <condition> at a certain level always points to the same identifier.

The JUMP-syntax:

```
<program> ::= <line>/<line><program>
<line> ::= <statement>/<label><statement>
<statement> ::= <action>/IF <condition> GOTO <label>
<action> ::= <simple action>/<simple action> AND <action>
<simple action> ::= A1/A2/A3/..../A10
<condition> ::= C1/C2/C3/..../C8
<label> ::= P1/P2/P3/..../P9
```

The basic symbols in both languages as well as the problem defining elements were meaningful words in the native language of the subjects. In order to prevent typing speed to bias any result we redefined the entries on the keyboard, so that all language elements could be produced by pressing single keys. By pressing a FORM-key the subject was able to rearrange the format of his text automatically, providing indentation in NEST, and label tabulation in JUMP. Another key was used to submit the text to a syntax check. In case the program was shown to be syntachically correct, it could be run for a semantic check by pressing a TEST-key. 63 subjects, young adults of different occuptions, all having completed secondary education, took part in the experiment. The programming session was preceeded by a test-session that supplied data for the composition of a profile of learnercharacteristics consisting of:

- the score on set II of Raven's Advanced Progressive matrices as an index of the general level of intelligence;
- the scores on the three learning style dimensions;
- information concerning the educational background. Here we classified our subjects in three categories:
  - betas : 21 subjects who (according to themselves) were rather good in math, and enjoyed it as well.
  - alphas: 22 subjects who weren't any good at math and didn't like it either.
  - ? : 20 subjects who reported to de well in math but did not enjoy it and those who weren't any good at it, but enjoyed it all the same.

On the basis of previous findings it was predicted that on the average the betas would make the best programmers, and the alfas would make the poorest ones. The two groups in the ? category, were supposed to perform somewhere in between. By making comparisons between test scores profiles we were able to assign two almost identical groups to both languages-conditions. It should be noted that our sample was rather heterogeneous.

#### the learning phase

At the programming session our subjects were introduced the apparatus, the procedure, and the secrets of programming by means of self-instruction. Sitting behind a terminal they followed instructions from a manual at a self paced rate. Each significant action was followed by immediate feedback on the screen. In order to stress the strict obedience expressed by the machine we used the metaphor of the computer as a modern, multi-purpose slave. In the first part of the programming session, the learning phase, our subjects could call for assistance from the experimenter whenever this was felt



necessary. This phase terminated after completion of two programs. These programs, borrowed from Sime et al (1977) were called Cook I and Cook II. To give an example, Cook II, more complex than Cook I, was specified by the following requirements:

"you have to program your slave in such a way that he will:

grill—everything that is not big and not leafy,  
peel and fry—everything that is leafy and not hard,  
chop and grill—everything that is not leafy but big,  
chop and boil—everything that is both leafy and hard".

#### results of the learning phase

In our heterogeneous sample it appeared that the scores on learning style factors 2 and 3 correlated very high ( $r = .68$ ). For this reason we decided to combine these two scores, and to call the composite result a versatility-score. Pask reserves this label for an opportunistic learning style that includes both operation learning and comprehension learning. Versatile students may choose at will between the two ways of structuring learning material. They have a 'complete' repertoire of materials both for constructing particular rules and for creating general descriptions. Our criterium for versatility in this case was a combined score on factors 2 and 3 that was above median. Analyses of covariance on learning performance, eliminating the effect of intelligence, showed that beta-subjects indeed needed less time than alphas. ( $F_{2,50} = 3.63$ ,  $p < 0.05$ ) respectively 75 against 104 min. The subjects we could not label alpha or beta needed 92 min. The analysis also showed that learning NEST takes less time than learning JUMP, respectively 79 and 99 min. ( $F_{1,50} = 5.28$ ,  $p < 0.05$ ). The final result of the analysis indicated that versatiles needed less time than non versatiles, 75 and 105 min. respectively ( $F_{1,50} = 4.21$ ,  $p < 0.05$ ). Looking at the three competence factors separately, it appeared that all had significant partial correlations with the learning time.

factor 1 inclination to learn	- .50
2 operation learning	- .40
3 comprehension learning	- .36

In this first stage, we didn't find any interaction between language type, learning style and educational background.

#### additional problems

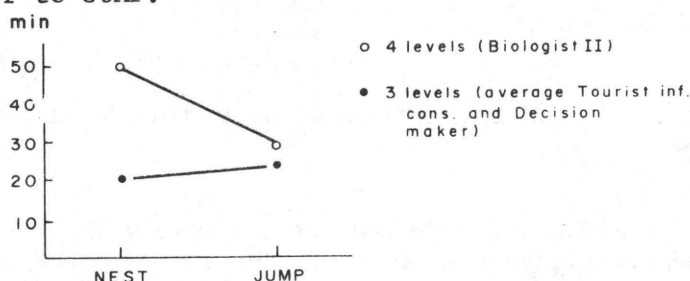
After the learning phase the students were confronted with four additional problems that had to be solved without any further help of the experimenter. Designing the experiment we argued that a comparison between language constructs should be accompanied by an analysis of the structural features of the algorithms to be coded in terms of these. The following variables were considered:

- a. The presence or absence of a hierarchy in the algorithm and the depth of embedding in such a hierarchy. We speak of hierarchy when a number of tests has to be carried out in some logical or preferred order. The 'depth of embedding' reflects the number of levels in this hierarchy.
- b. The presence or absence of a semantic framework that suggests some kind of (hierarchical) structuring. Most algorithms are conceived in terms of meaningful identifiers which imply a tacit reference to some semantic framework. This framework may in itself suggest a certain order of tests. When it is absent (e.g. in the case of meaningless identifiers) such order only follows from the formal properties of the problem structure.

Biologist I is the label for a task to program a taxonomy identifying animals (fish, worms, mammals etc.) by the presence or absence of certain unique features (feathers, fins etc.) The presence of one feature excludes the presence of all others and is a sufficient condition to identify the animal. Therefore there is no preferred or logical order of tests, and the depth of embedding is zero (that is, from the algorithmic point of view). The other three problems are hierarchical in nature. Tourist Information Consultant is a slave who advises tourists about means of transport (taxi, bus etc.) on the basis of data about destination (far or near), luggage (yes or no), budget (high or low) and time available (in hurry or not). Decision Maker is formally identical to the former algorithm, but lacks any semantic frame of reference, the antecedents being replaced by letters, and the consequents being replaced by Action 1, Action 2 etc. Biologist II shares the semantic domain of Biologist I, but the decisions, more realistic, have to be made on the basis of combinations of attributes which are not exclusive. Only a few of the 'possible' combinations point to existing categories of animals. There are four levels of embedding in this hierarchical taxonomy, as against three such levels in the preceding algorithms.

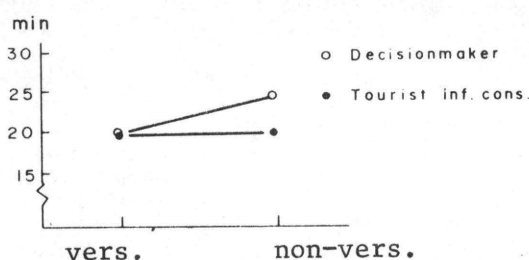
Results with the additional problems

In solving the additional problems 8 subjects dropped out, failing to complete their task within a reasonable amount of time. None of them was a 'beta', and only one of them was a 'versatile'. Motivation was not the reason to fail here. Although the experimenter suggested to leave at the end of the time agreed on (three hours) five of these subjects continued the experiment for one or two hours without extra payment. We omitted the results of these eight subjects from the rest of the analysis. The differences between the solution times for the problem with four levels (Biologist II) and the average results on the two problems with three levels (Tourist Information Consultant and Decision Maker) shows a significant interaction with type of language ( $F_{1,41}=4.98, p < 0.05$ ). This indeed suggests that some critical value is passed as the number of levels increases from three to four. With more levels NEST is inferior to JUMP.



There is yet another case in which JUMP is superior to NEST. For the non-hierarchical problem, Biologist I, NEST takes significantly more time (38 min.) than JUMP (16 min.) ( $F_{1,41}=5.59, p < 0.05$ ). Only in this case NEST invokes more syntax errors than JUMP (72% against 37% in first runs). First, one is apt to skip the IF NOT clauses in between the IF's since they seem superfluous in this case. Second, NEST, as opposed to JUMP, does not tolerate any redundant testing. Redundant tests induce syntactical omissions since they leave nothing to be specified in the IF NOT clause.

The meaningful problem Tourist Information Consultant takes less time than the subsequent Decision Maker, that is formally identical, also find a significant interaction with versatility ( $F_{1,41}=4.54, p < 0.05$ )



Although the abstract problem followed after the meaningful equivalent, non-versatiles met with considerable difficulties. It seems that the availability of semantic

but framed in abstract terms ( $F_{1,41}=9.94, p < 0.05$ ). Here we

connotations is indispensable to them in order to cope with a complex, hierarchical structure. As the semantics are removed, their solution strategies are inferior to those of versatiles. Finally we note some interesting results with learning style factor I, inclination to learn. Controlling for effects of intelligence, the partial correlations with the solution times for the Tourist Information Consultant and the complex algorithm Biologis II are  $-.39$  and  $-.52$  respectively. In the case of the abstract Decision Maker this correlation doesn't reach significance. Here it is the versatility-style that is predictive for success.

#### FUTURE RESEARCH

At the moment we are preparing a series of experiments as a follow up for the one reported here. In designing these, we will take the following considerations into account:

##### embedding

It appears that NEST is inferior to JUMP when there are too many levels of embedding or when the algorithm proper doesn't ask for a hierarchy at all. Both difficulties can supposedly be dealt with by the introduction of another branching alternative presented by Atkinson (1979) as NEST-PA (positive alternatives). According to Atkinson a programmer should be allowed to 'think positively' by avoiding negative formulations. With the help of this construction, which looks like the CASE-statement in PASCAL, we can cope both with adjacent conditions (Biologist I) and with hierarchical tests. We will compare this alternative with NEST-INE.

##### didactics

The present results suggest that we should not exceedingly dramatize the differences between languages or language features. Whether or not somebody will be successful in performing a programming task only partly depend on the qualities of the syntax. Much will depend on the combinations of problem characteristics, and individual cognitive style factors or problem solving strategies. Furthermore, a language that looks very hard to learn, nevertheless may easily map into the learner's experience. In our future experiments we will study the use of structural diagrams as a didactical aid for the understanding of flow of control principles.

##### "Tie-structures

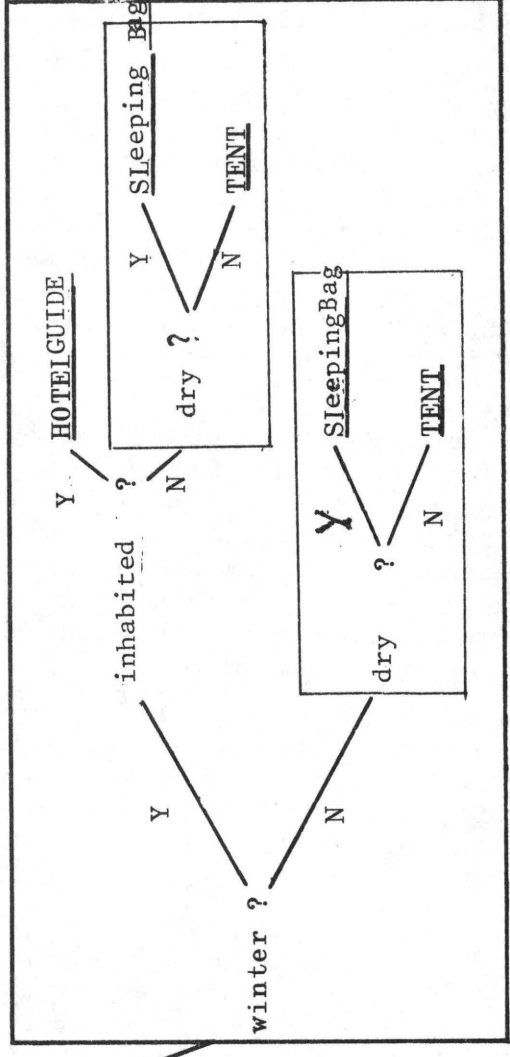
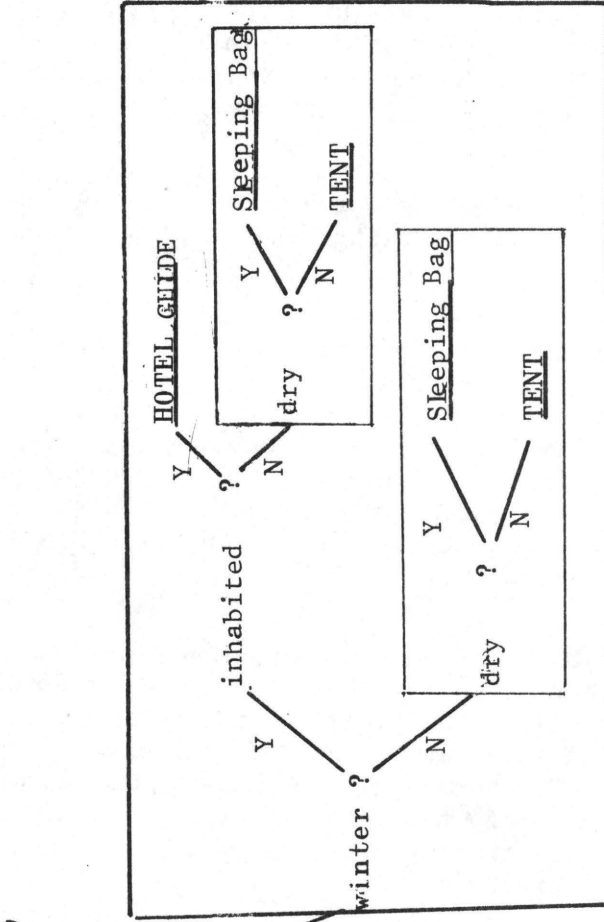
Until now we have not paid any special attention to problems, in which some part-structure is referred to from several places in the algorithm, while there may be no question of a hierarchy whatsoever.

Going on holiday, take with you:

- a caravan : if driving licence and same continent;
- a hotelguide : if no driving licence or not same continent,  
if winter and to inhabited area;
- a tent : if no driving licence or not same continent,  
if summer or uninhabited area, and not dry;
- a sleeping bag : if no driving licence or not same continent,  
if summer or uninhabited area, and dry.

Such problems that we tentatively call "tie"-problems ask for special modularisation devices. We don't know yet what precise form they will take, but our first prototype will be parameterfree.

CARAVAN



driving licence?

Y

N

Y

N

Y

N

same continent?

winter?

inhabited

HOTEL GUIDE

Sleeping Bag

TENT

Y

N

Sleeping Bag

TENT

Y

N

dry

dry

dry

dry

dry

HOTEL GUIDE

Sleeping Bag

TENT

Y

N

dry

inhabited

Y

N

dry

dry

Sleeping Bag

TENT

Y

N

dry

dry

dry

dry

## 7. REFERENCES

- Atkinson, L.V. Should if...then...else...follow the dodo? Software-Practise and Experience, 9, 693-700 (1979).
- Green, T.R.G., Sime, M.E. & Fitter, M.J. The art of notation. In: Computing skills and the user interface. Edited by Coombs, M.J. & Alty, J.L. Academic Press, London, 1981.
- Guilford, J.P. & Hoepfner, R. . The analyses of intelligence. McGraw-Hill, New York, 1971.
- Papert, S. Mindstorms. The Harvester Press Ltd., Brighton, 1980.
- Pask, G. Styles and strategies of learning. British Journal of Educational Psychology, 1976, vol. 46, 128-148.
- Seidman, R.H. The effects of learning the logo computer programming language on conditional reasoning in school children. University Microfilms International, Ann Arbor, 1982.
- Sime, M.E., Green, F.R.G. & Guest, D.J. Scope marking in computer conditionals - a psychological evaluation. International Journal of Man-Machine Studies, 1977,9, 107-118.
- Soloway, E., Lockhead, J. & Clement, J. Does computer programming enhance problem-solving ability? Some positive evidence on algebra word problems. In: Computer Literacy, edited by Seidel, R.J., Anderson, R.E. & Hunter, B. Academic Press; New York, 1982.
- Van der Veer, G.C. & Ottevangers, D.C. Problem solving by programming. Proceedings of the Digital Equipment Users Society, 1976, vol. 3, no. 1, 345-351.
- Van der Veer, G.C. & van de Wolde, J. De leerstijlen van Gordon Pask. Een Nederlandse bewerking van de Smokkelaarstest. In: Strategieën in leren en ontwikkeling, edited by Lodewijk, J.G.L.C. & Simons, P.R.J. Swets en Zeitlinger, Lisse, 1982.



A model of the programmer's abilities to understand program semantics and its impact on program(ming language) design

H.-E. Sengler  
URW-Unternehmensberatung, Hamburg

## 1. Introduction

This paper suggests a model describing the understanding of a computer program by an experienced programmer. Its purpose is:

- (1) to offer an explanation of the problems programmers experience when managing large programs and
- (2) to suggest improvements in the design and construction of programs in order to reduce these problems.

The model does not discuss problems of learning a programming language or learning a computing system but assumes that the programmer is well acquainted with both.

The understanding of a program is seen as a process of acquiring information, from the outside world as well as from information already available. Two characteristics of human thinking are taken into account:

- (1) storage and retrieval of information is done by association
- (2) the amount of information processable at any one time is limited

## 2. Premises

The source of information for a programmer is the program's source text, accompanied possibly by additional descriptions of special aspects of the program, in general being one or more sections of text or graphics. Each such section may contain formal and informal passages. In the context of this model the formal passages are those that determine the effect of the program (i.e. the behaviour of a computer that executes the program) whereas the informal passages do not contribute to that effect.

The model excludes the understanding of the informal passages of a program because:

## A Model of Understanding a Program

- (1) The information gainable from the informal passages is in general vague and its contribution to the understanding of a program therefore difficult to assess.
- (2) The programmer can not rely on the informal passages because they often do not exist at all or are out of date.
- (3) The production of informal passages for programmers is an additional effort since they do not contribute to the programs effect. It would reduce the overall costs of constructing a program if a programmer could understand it solely from its formal passages.

The aim of understanding a program, i.e. the information a programmer wishes to obtain about it, depends on the purpose for which it is analysed. The purpose the model assumes is to alter the effect of a program in order to achieve some desired new effect. The aim of understanding can be described as the "semantics of the program", defined as "the effect, the program will cause when being executed by a computer and the correspondence between passages of the effect and the passages of the program causing them".

The formal passages are written in a language whose syntax and semantics are well-known to the programmer. The semantics of the program is completely defined by its formal passages. The model thus describes the derivation of a program from the formulation of its formal passages.

### 3. The internal program

A human, reading a text in a language familiar to him does not analyze it characterwise but reads words and makes assumptions about the words to follow. A corresponding behaviour is assumed for a programmer reading a program. It can be interpreted as mapping the text into a new structure formed within the human brain that represents all aspects of the text the reader assumes to be essential. The structure formed when reading a program will be called an "internal program". All subsequent processes of understanding described in the model are based on the internal program.

The properties of an internal program are as follows: An internal program is a net. Its nodes are called "components", its arcs are called "relations". Basically the components of an internal program are the semantic units of the programming language as they are used in the program, the relations accordingly are the semantic relations between them.



## A Model of Understanding a Program

For example (all examples are from PASCAL) the character "+" will be mapped into a component with the semantics "add the left and right operands (when execution reaches this point)" with relations to both operands and the previous and following operation. The character "w" in the world "while" will not be mapped into a component, nor would the world "while" alone because only for the combination "while"... "do" the semantics is defined.

The actual mapping of a program text into an internal program however is not determined by the language alone but is additionally formed by the individual experiences a programmer makes while learning the language and while using it. Attempting to reduce the amount of information he has to cope with, the programmer may interpret subnets of the internal program (as they would appear according to the semantics of the language) as single components if they often occur in the same combination.

A subnet mapping may be influenced by the kind of programs the programmer works with but it may also be suggested by the language. For example a variable, its type and the read- or write-accesses to it are separate components according to the semantics of the language yet a programmer may map them into a single component "variable" with a property (type) and relations to operations (accesses).

As a result one cannot assume one unique mapping of the formal passages of a program into an internal program but rather a variety of similar mappings.

### 4. The process of understanding

When analyzing a program the programmer is assumed to look at the program text and at each moment concentrate on one portion of it. This portion is mapped into the corresponding portion of the internal program (cum grano salis, as he may have to search for or remember information not within the current view). This portion of the internal program is information that is directly accessible for further processing.

In the choice of what parts of the program text to regard as portions the programmer is guided by the appearance of the program. Its formal passages are usually separated syntactically into "regions". Examples of such regions are blocks ("begin"... "end") or lines of text enclosed by blank lines. The programmer will choose a region as a portion unless no such separation exists or a region contains too much information to be processed. In that case he will define his own separation.

## A Model of Understanding a Program

A portion of the internal program may contain different kinds of information: Components and relations whose semantics are known to the programmer, outer relations (i.e. relations to components or regions outside the currently analyzed portion) and inner portions (i.e. portions inside the currently analyzed portion) whose semantics may as yet be unknown. (Note: this implies that a programmer separates a program into portions that are either within each other or are mutually exclusive).

Before being able to conclude the semantics of a portion the programmer must acquire (i.e. understand) the semantics of the outer relations and inner portions that he doesn't know yet (if there are any). This forces him to leave the current portion and concentrate on other portions of the program. As a result he has to memorize those semantics he has already understood and remember them after he returned to the portion he started with.

After understanding and remembering all semantics of outer relations and inner portions the programmer can conclude the resulting semantics of the current portion. As this semantics in general is too complex to allow further processing he additionally has to abstract it, i.e. develop a concept of the functional qualities of the portion or of the subproblem the portion solves within the surrounding program.

This description of the understanding of one portion covers implicitly that of its inner portions. By viewing a program as a portion too, the whole process of understanding a program can be described in the following grammar-like form:

Understanding a program = understanding a portion.

Understanding a portion =  
Finding its components, relations, outer relations and inner portions;  
Understanding the outer relations and inner portions;  
Associating the semantics of components and relations;  
Remembering the semantics of outer relations and inner portions;  
Concluding the resulting semantics of a portion;  
Abstracting the resulting semantics of a portion.

Understanding an outer relation =  
Finding the portion connected to it;  
Understanding the portion connected to it;  
Concluding the semantics of the relation;  
Abstracting the semantics of the relation.

## A Model of Understanding a Program

The processes of finding, associating, remembering, concluding and abstracting are assumed to be basic, having the following characteristics:

- a) Finding (assumed to be done with the eye) is easy if symbols representing components, relations and boundaries of regions are clearly visible and distinguishable and if relations to components outside the current view are represented with an indication where to find the related component.
- b) Associating is easy if the semantics of components and relations of the language are distinguishable from each other and if there is a correspondence between kinds of semantics and kinds of their symbolic representations. If the language offers a large number of components and relations they should be ordered hierarchically.
- c) Remembering is easy if there are hints that identify or characterize the semantics to remember and if the semantics is not splitted into loosely coupled parts.
- d) Concluding is easy if the number of details to include is low.
- e) Abstracting is easy only if there are clear indications as to what concept the resulting semantics of a region represents.

Note: As an interesting result from the characterization of the processes of remembering and abstracting one finds that not only does the information from the formal passages not suffice and the informal passages are needed as well but also what information the informal passages should contain.

## 6. Explanations

One purpose of the model is to offer explanations for the problems programmers encounter when managing large programs. Some examples of explanations of well-known problems are:

- (1) Side-effects: In terms of the model they are relations which are not visible and therefore hard to find.
- (2) Self-modification: Increases the number of details that have to be included for the conclusion of the resulting semantics, means a misleading correspondence between the symbol in the program text and its semantics.
- (3) Recursion: Hinders the conclusion at the lowest level of portions because a not fully understood semantics has to be included. Yet: offers a concept for abstraction.

## A Model of Understanding a Program

Explanations why certain programming techniques yield "good" programs can be offered as well, e.g.:

- (4) Stepwise refinement: The model neatly coincides with that method.
- (5) One-in, one-out: Reduces the number of relations of a region, supports abstraction by suggesting the concept "(sub)action".
- (6) Abstract data types: Reduces the number of relations to variables, offers a concept supporting abstraction.

### 7. Useful program properties

Besides the explanation of problems the purpose of the model is to suggest improvements in program design. If the assumptions of the model are valid the following suggestions should enhance program understandability:

- a) A program should be hierarchically divided into regions, each region containing only few details (i.e. few components, relations, inner regions and outer relations).
- b) All components and relations should be represented (e.g. no implicit declarations). As far as possible all information determining the semantics of a region should be represented within that region. Global definitions could be displayed within the region by interactive workstations.
- c) The area in which the components of a region lie as well as its outer relations should be represented graphically. Lines could represent relations, shaded areas could represent regions, colours could distinguish different kinds of components or relations.
- d) There should be few elements in the programming language. If there have to be many they should be ordered hierarchically. If it is obvious that combinations of elements will be understood as components the combinations should be defined as elements. The representations of the elements should be chosen to correspond to the kinds of elements.
- e) The language should offer different kinds of abstraction mechanisms to serve as hints to remember the semantics of a region. The programmer should give hints for remembering the semantics, e.g. by carefully choosing the names.
- f) The machine the programmer has to imagine when concluding the resulting semantics of a region should have few states or there should be no machine to imagine at all.
- g) The programmer should give for each region a description of its functional qualities and/or the subproblem that is solved with it.

## A Model of Understanding a Program

### 8. Final remarks

The author apologizes for not giving any references so far. This is due firstly to the limited space available, secondly to the inability of the author to trace back all origins of the ideas based on. The most influential works are listed below.

The reader will have found that many of the recommended program properties cannot be realized by methods of program construction alone but must be supported by programming language design and programming system design. To exemplify his objectives the author has defined a programming language (see below) using graphics as the means of representation and with a group of colleagues has implemented a first version of it on a PDP11 computer. The project is supported by the West German government (BMFT/GMD) under grant no. 083 0214.

### References

- E.W.Dijkstra           The humble programmer  
CACM 15 (1972) 10, 859-866
- E.W.Dijkstra           On the teaching of programming, i.e. on the  
teaching of thinking  
In : F.L.Bauer, K.Samelson (eds.)  
Language hierarchies and interfaces  
LNCS 46 , Springer Berlin 1976
- T.R.G.Green            (Various contributions on the understandability  
of language elements as well as on graphical  
representations)
- C.A.R.Hoare            Notes on data structuring  
In : O.J.Dahl et al.  
Structured programming  
Academic Press London 1972
- H.-E.Sengler           Programmieren mit graphischen Mitteln: Die  
Sprache GRADE und ihre Implementation  
In : H.Woessner(ed.)  
Programmiersprachen und Programmentwicklung  
IFB 53 , Springer Berlin 1982
- N.Wirth                Program development by stepwise refinement  
CACM 14 (1971) 4 , 221-227
- N.Wirth                Programming languages: what to demand and how to  
assess them  
Report 17 ETH Zuerich, Institut fuer Informatik,  
March 1976



ANALYSIS OF BEGINNERS' PROBLEM-SOLVING STRATEGIES IN PROGRAMMING

---

Jean-Michel HOC

C.N.R.S. (Paris, France)

INTRODUCTION

During the last decade, ergonomical studies on programming have been widely developed, as one can see in recent syntheses of works carried out in the field (Smith & Green, 1980 ; Shneiderman, 1980). But these studies are also much criticized : a methodological debate has recently been opened on this problem (Sheil, 1981 ; Moher & Schneider, 1982). These studies are criticized with being at the same time too much directed by computer technology and also not giving enough methodological precision.

It is true that the theoretical frameworks of cognitive psychology do not permit us to derive very precise models of the psychological mechanisms underlying programming. But these models are, without doubt, richer than the implicit frameworks which have directed a lot of studies on programming. Models are necessary to base the relevance of the experiments. When they cannot be entirely derived from a theoretical approach, the role played by relatively open observation of behavior, in order to obtain missing information, must not be neglected. Indeed, it is difficult to generalize the results of such observational studies, but they may lead to more relevant and economical experiments, the aim of which is to be inductive.

This methodological option has directed my own works, as well as those of authors such as Brooks(1977). Brooks tried to modelize programming strategies at an expert level, while my contribution related to beginners' strategies, with application to perspectives of training. The study of such strategies was necessary for a better understanding of the difficulties met in the acquisition of top-down programming methods.

I shall restrict myself to reporting the main results of my own works, published in french, hence less attainable. They are placed between two studies, published in english, and about which I shall not speak : an exploratory study of strategies at diverse levels of expertise (Hoc, 1977) and a comparative assessment of two top-down programming methods (Hoc, 1981).

At first, I shall present a longitudinal and observational study (Hoc, 1978a) of a training course for a business programming method, widely used in France : the L.C.P.\* method (Warnier, 1975). From it, I shall extract the two essential

---

(\*) "Logique de Construction des Programmes"

research topics which directed the design of subsequent experiments : the learning of the computer operation and the mechanisms in expressing procedures.

### I. A STUDY OF A TRAINING COURSE FOR A TOP-DOWN PROGRAMMING METHOD (Figure 1)

Program design strategies are not innate ! They are learned by training and professional practice. Training is not confined to learning programming languages, as was the case in the past : now programming methods are taught. In France, these methods are in a large part inspired from structured programming. We tried to assess the learning of one of them : the L.C.P. method (Hoc, 1978a). Afterwards an assessment of the same type was done for another method (the Deductive method of Pair, 1979), by a colleague (Kolmayer, 1979), and led to similar psychological conclusions.

The L.C.P. method is a business programming one which consists at first in structuring the results and the data, and then in "deducing" (according to Warnier, its designer) the program structure. Each structure is constructed by a top-down (planning) method, in embedding and chaining two basic constructions : the conditional structure and the iterative structure.

In an intensive training centre, I have for three months done a longitudinal analysis of the acquisition of this method. I studied the structural compatibility between the steps (results, data, program, and flow-chart), and their correctness (error analysis).

Here I shall only cite the two main results :

(a) The structural compatibility is not very good. Even at the end of training, for each step, the subject does a new analysis of the problem, without clear coordination with the preceding step. There is no "deduction" of the program structure from the data structure.

(b) The performance improves when constructing the flow-chart of the program. The preceding steps often contain errors. At the beginning of the training course these structure errors mainly come from data representations incompatible with the computer operation (for example : data access mode). These errors are corrected when writing the flow-chart.

I interpreted these results by setting up two working hypotheses :

(1) The structure of the data and the results, hence their representations, are not independant from the processes the subject aims to apply to them. Some false structures might be correct relating to processing devices well known to the subject. Here, I shall return to the idea (Hoc, 1977) according to which



the beginner uses representation and processing systems\* based upon devices other than the computer, before having constructed an appropriate new system by differentiation. A characteristic of beginners' strategies is perhaps a mechanism of adapting known procedures to the computer operation.

(2) The improvement of the structure in writing the flow-chart might lead one to think that the beginner could not adapt a procedure without access to the paths the machine will go up during the execution. Another characteristic of beginners' strategies could be seen as generating the instructions in a situation of mental execution of the program.

Following these observations, I did experiments in order to study these hypotheses more closely.

## II. LEARNING THE COMPUTER OPERATION

I carried out two experiments successively with beginners according to a common paradigm. A problem is used for which it is sure the subject knows a procedure. At first this procedure is examined in all its details, afterwards the subject is placed in a situation where, step by step, he commands a computer device simulated by a VDT. Constraints in the machine operation are gradually introduced. This paradigm has the aim, for the beginner, of right away introducing the knowledge of results which is deferred in the usual programming situation (Hoc & Kerguelen, 1980). Mainly the errors and the response latencies are analysed.

### II.1. An updating stock problem (Hoc, 1978b : Figure 2)

An Old Stock file must be updated from a Transaction file. A computer reduced to four operations is defined : inputting an Old Stock record, a Transaction one, adding the two quantities, and writing a New Stock record. In the first situation the subject can see the data. Afterwards he cannot do so any longer. Finally he must do the task blindly, by commanding tests.

The results can be summed up in the following way :

(a) Very quickly the subject elaborates a sequence which I see as typical. This sequence, valid in processing an Old Stock record with one transaction, will be transferred (or its overall structure), and adapted in order to process the records of another type (without transaction and with several transactions). This transfer persists up to the last situation.

---

(\*) "Systèmes de Représentation et de Traitement"

(b) In the first situation, the subject concentrates on identifications just before the input of an Old Stock record (he counts the transactions to be processed, and afterwards he links up the operations very quickly). In subsequent situations, the typical sequence must be broken, because identifications can only be done in the memory cells (transaction by transaction). The subject then operates less in terms of systematic identifications than of hypotheses on the number of transactions by record, in order to avoid breaking up the structure of the typical sequence. If he accepts to see the hypotheses invalidated (error messages), he refuses to test these hypotheses by himself. This mechanism is still much used in the last situation no matter in what way the subject is invited to command these tests explicitly.

## II.2. A sorting problem (Nguyen-Xuan & Hoc, 1981)

The same experimental paradigm was used for a more complex problem : sorting. I can only evoke the results of this experiment which clearly shows the limits of learning by doing, concerning the computer operation.

Here, the command situation was less constraining. Several procedures were possible. If the subject clearly transfers his usual procedure (often an inserting procedure), the adaptation rarely led to an optimal procedure taking the best advantage of the machine operation. This adaptation was directed by a regulation of the mental load (especially the memory load) :

- the subject stops the adaptation as soon as the mental load is judged tolerable ;
- the errors provoked by an overload are not sufficiently analysed, and the subject changes his procedure without these changes giving any improvement.

Learning by doing seems to be of interest only with constraining devices which direct the subject towards an optimal procedure. It is of little interest when the number of alternatives is too large.

## III. EXPRESSING THE PROCEDURES

At the end of the two experiments cited the subject was asked to express the procedure which he had just used and which was correct. In the first case (updating), he had to write a flow-chart, structured or not, in the second case (sorting) he wrote up the procedure in natural language. The aim was to examine formal and verbal reports in situations where the processing device was well-

defined and the procedure yet elaborated.

### III. Flow-charting (updating)

After having perfected a correct procedure of updating Old Stock records with 0, 1 or several transactions, the subject had to construct three flow-charts in that order : the case where the records have one transaction, the case where they have either 0, or one transaction, and the general case.

A group of subjects (prescribed planning) was constrained to

- use only the two basic constructions of structured programming,
- and, apart the first one, generate the flow-charts in a top-down mode, by transferring the overall structure (plan) of the preceding flow-chart.

The other group (free expression) had no particular constraint.

Here I shall present only the three principal results (Hoc, 1979) :

(a) In free expression, the subject expresses the flow-charts in following the order of an execution : that is the mental execution of the program which is directing the expression and not a representation of the program structure which the subject discovers only at the end.

(b) In prescribed planning, the order in which the instructions are written notably deviates from the order of execution, but the subject refers to the production rules (condition - sequence of actions) used in the execution, by fitting them to the prescribed basic constructions.

(c) Concerning the gross performance (time : Figure 3), in prescribed planning, we can see that, if these fitting difficulties are embarrassing for the subject at the beginning, this handicap disappears as soon as the method become more familiar and the problem more complex. In free expression, the subject is far too much affected by its increasing complexity.

#### III.2. Expression in natural language

The analysis of the free expression of sorting procedures in natural language led to the following results :

(1) Before expressing the structure of the procedure, the subject for a long time expresses particular executions of this procedure. This phenomenon is more especially obvious when the subject has attained a procedure with the principle of which he is not familiar.

(2) The top-down mode of expressing the control structure is more often used for the sub-structures than for the super-structures.

(3) The conditional constructions are more difficult to express than the iterative ones.

## CONCLUSION

These experiments bear out the hypotheses set up after the field study, but they show a larger complexity in the beginners' strategies. It is true that they are characterized by adapting known procedures to the computer operation, and by a mental execution of the program. But we have also seen that :

- if this adaptation can be facilitated by command situations where the knowledge of results is without delay, these situations must be sufficiently constraining so that the subject took the best advantage of the machine operation,
- we can see strategies of progressive generalization of a sequential procedure,
- it is necessary to express an execution of the procedure (even if it is already elaborated in a command situation), before becoming aware of its control structure,
- finally, the conditional structures are difficult to express, possibly when the corresponding tests are not explicitly performed in the execution (other more complex control structures are possible).

I guess that a too early training of top-down programming methods should be avoided, before the subject has learned the constraints of the computer operation on the overall structure (plan) of the procedures and knows a wide range of concrete plans to be transferred. It has not been proved that these top-down methods are always feasible even at an expert level.

I have begun a research program with the aim of characterizing the components of expert's strategies and their conditions of implementation, and in particular top-down strategies. The practical problem is to contribute to the design of computerized aids in programming. The results of this research may permit a better setting up of training problems.

## REFERENCES

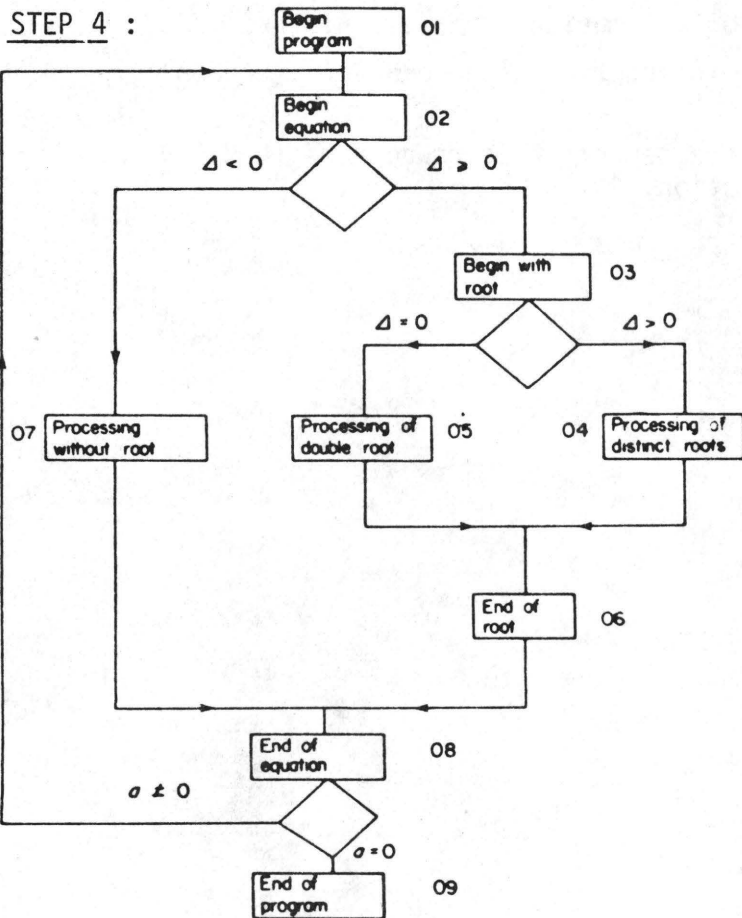
- Brooks, R. : Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 1977, 9(6), 737-751.
- Hoc, J.M. : Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies*, 1977, 9(1), 87-105.
- Hoc, J.M. : Etude de la formation à une méthode de programmation informatique. *Le Travail Humain*, 1978<sup>a</sup>, 41(1), 111-126.
- Hoc, J.M. : La programmation informatique comme situation de résolution de problème. Thèse de 3ème Cycle, Paris, Université René-Descartes, 1978<sup>b</sup>.

- Hoc, J.M. : Le problème de la planification dans la construction d'un programme informatique. *Le Travail Humain*, 1979, 42(2), 245-260.
- Hoc, J.M. : Planning and direction of problem-solving in structured programming : an empirical comparison between two methods. *International Journal of Man-Machine Studies*, 1981, 15(4), 363-383.
- Hoc, J.M., Kerguelen, A. : Un exemple de dispositif informatique expérimental pour la psycho-pédagogie de la programmation. *Informatique et Sciences Humaines*, 1980, 44, 67-86.
- Kolmayer, E. : Développement et évaluation d'une méthode de programmation. Nancy, C.R.I.N., 1979, 79R074.
- Moher, T., Schneider, G.M. : Methodology and experimental research in software engineering. *International Journal of Man-Machine Studies*, 1982, 16(1), 65-87.
- Nguyen-Xuan, A., Hoc, J.M. : Adaptation d'une procédure connue aux règles de fonctionnement d'un ordinateur : la sériation. Paris, E.P.H.E., 1981.
- Pair, C. : La construction des programmes. *R.A.I.R.O.-informatique*, 1979, 13(2), 113-137.
- Sheil, B.A. : The psychological study of programming. *Computing Surveys*, 1981, 13(1), 101-120.
- Shneiderman, B. : *Software Psychology*. Cambridge, Mass. : Winthrop, 1980.
- Smith, H.T., Green, T.R.G. : *Human interaction with computers*. London : Academic Press, 1980.
- Warnier, J.D. : *Les procédures de traitement et leurs données (L.C.P.)*. Paris : Les Editions d'Organisation, 1975.

STEP 1 : Output file { result related to an equation (E times) } { existence of root(s) (0 or 1 time) } { existence of root(s) (0 or 1 time) } { distinct (0 or 1 time) } { distinct (0 or 1 time) } { x' (1 time) } { x'' (1 time) } { x (1 time) } { text: "no real root" (1 time) }

STEP 2 : { Input file { equation (E times) } { a, b, c (1 time) } } { Run file {  $\Delta \geq 0$  (0 or 1 time) } {  $\Delta < 0$  (0 or 1 time) } } {  $\Delta > 0$  (0 or 1 time) } {  $\Delta = 0$  (0 or 1 time) }

STEP 3 : Unit of processing { begin program (1 time) } { processing an equation (E times) } { end of program (1 time) } { begin equation (1 time) } { processing with root(s) (0 or 1 time) } { processing without roots (0 or 1 time) } { end of equation (1 time) } { begin root(s) (1 time) } { processing of distinct roots (0 or 1 time) } { processing of double root (0 or 1 time) } { end of root(s) (1 time) }



STEP 5 :

01. read  $a, b, c$
02.  $\Delta = b^2 - 4ac$   
if  $\Delta < 0$  go to 07
03. if  $\Delta = 0$  go to 05
04.  $x' = (-b + \sqrt{\Delta})/2a$   
 $x'' = (-b - \sqrt{\Delta})/2a$   
write  $x, x''$   
go to 06
05.  $x = -b/2a$   
write  $x$   
go to 08
06. write "no real root"
07. read  $a, b, c$   
if  $a \neq 0$  go to 02
08. stop.
- 09.

Figure 1 : L.C.P. analysis of second degree equations processing.

The five steps : results, data, program, flow-chart, detailed instructions (following Hoc, 1981)

OLD STOCK FILE

TRANSACTION FILE

NEW STOCK FILE

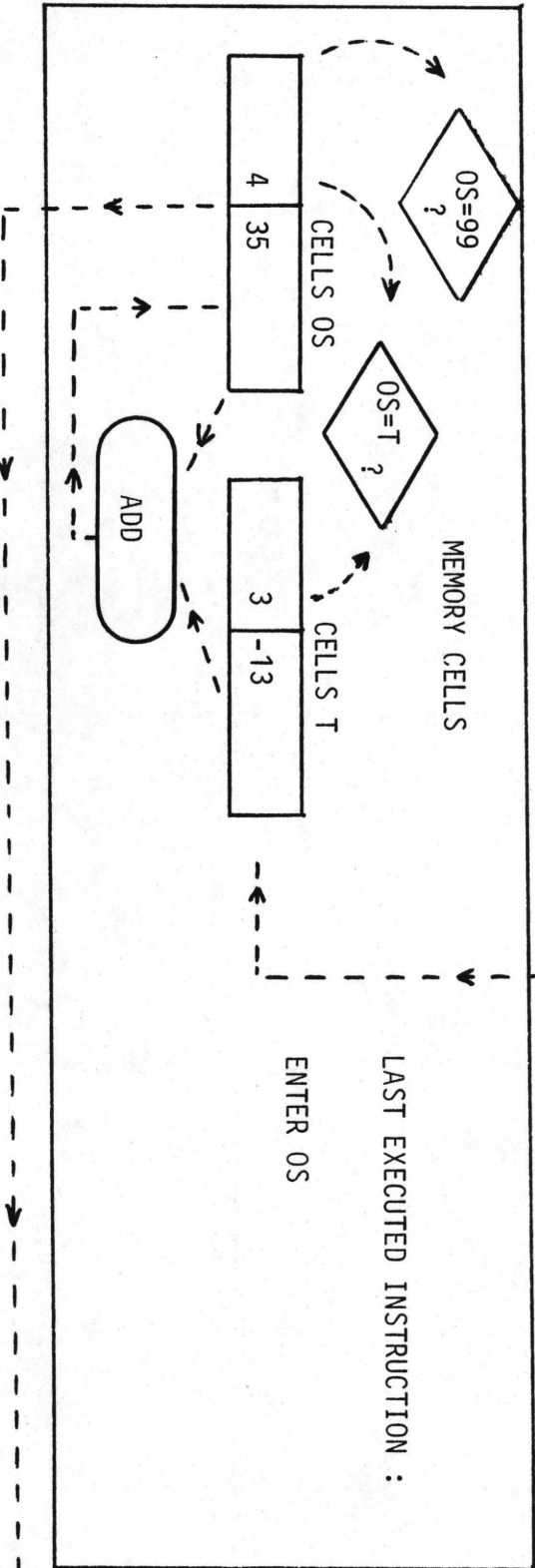
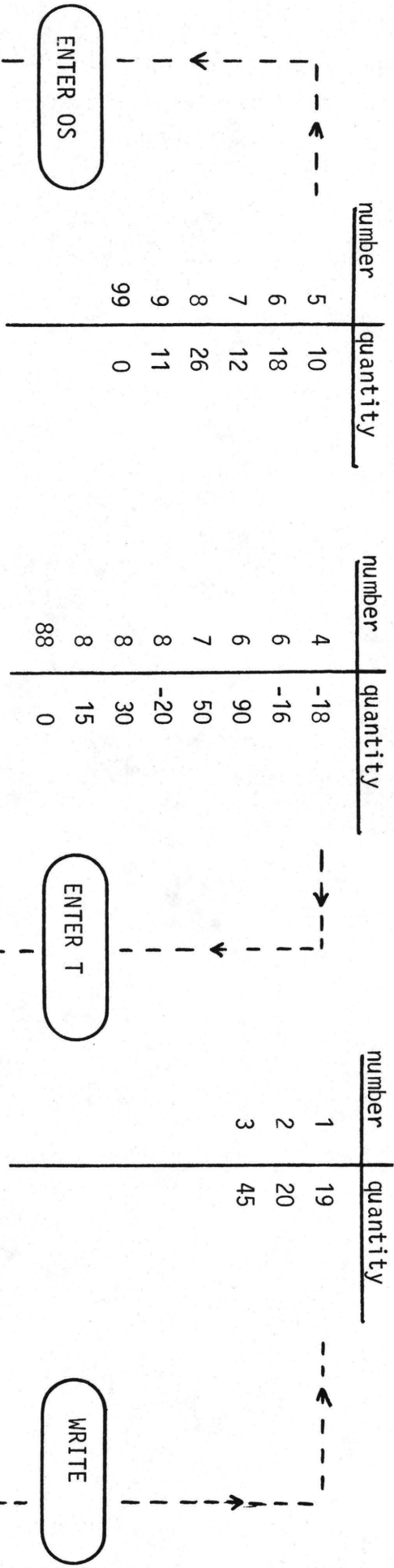
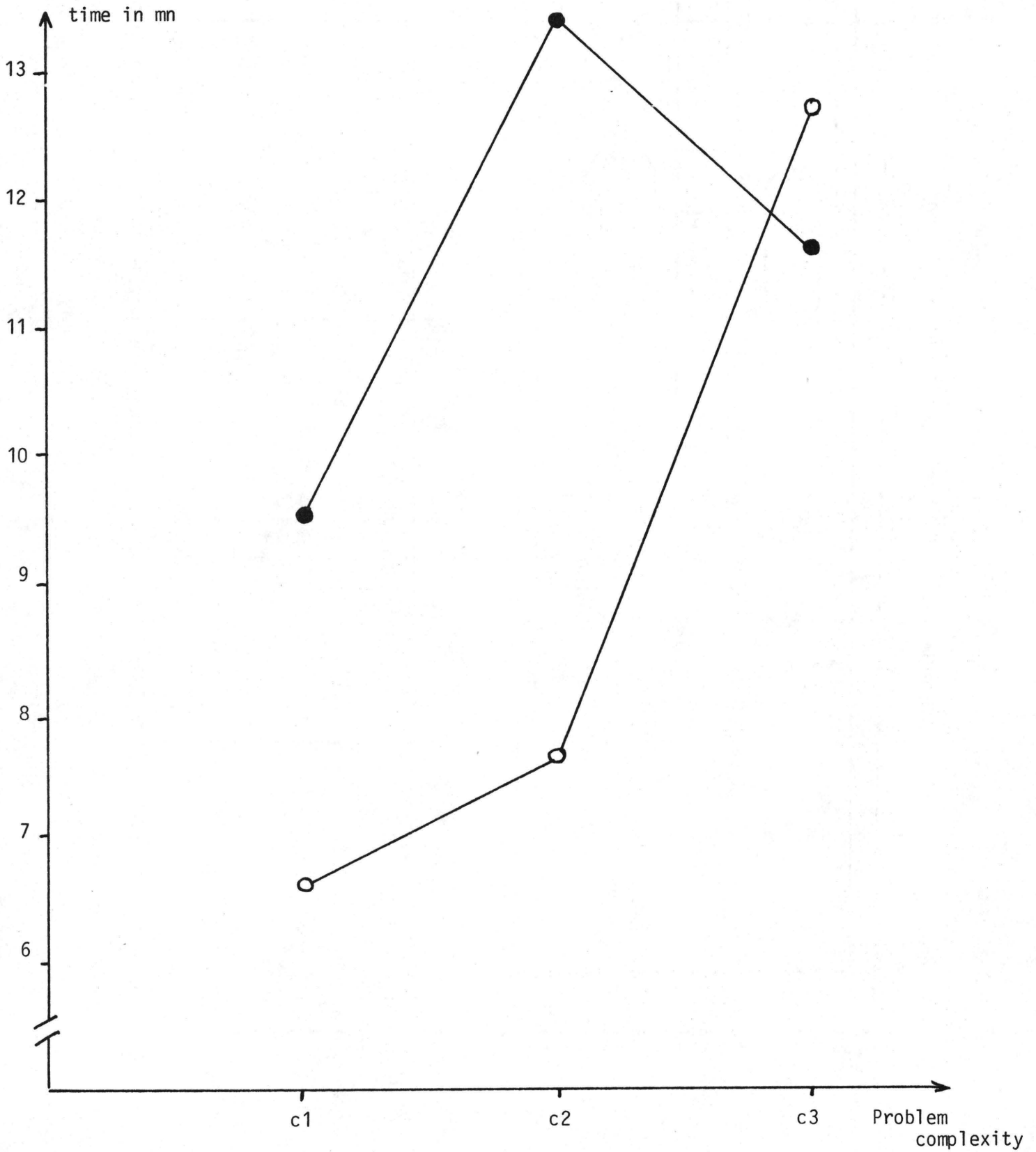


Figure 2 : Updating device with the four operations and the two tests (following Hoc, 1978b)

Figure 3 : Effect of a top-down method in writing a flowchart with increasing problem complexity (following Hoc, 1978b)

- free expression
- prescribed planning





PROBLEM SOLVING BY NOVICE PROGRAMMERS

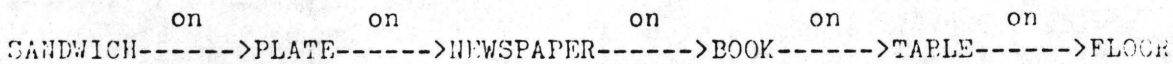
Hank Kahney  
Open University  
Milton Keynes, England

1) INTRODUCTION.

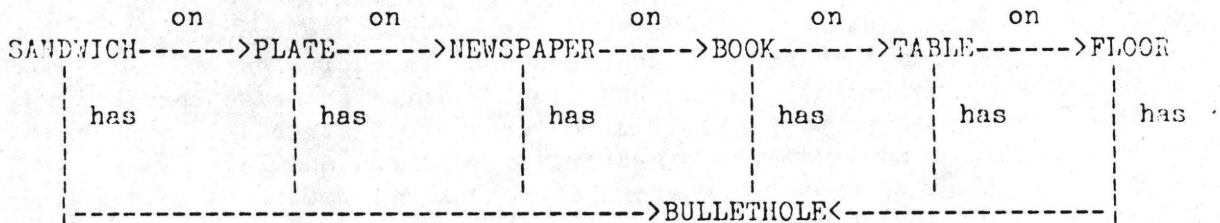
In the pages that follow we present a model of the behavior of novices who are learning artificial intelligence programming. Our novices are Open University students taking a third level course in Cognitive Psychology. As this course is favorably oriented to computer models of cognitive processes, the students are given a course in artificial intelligence programming early in the academic year. Their programming is self-taught from a Programming Manual, although they are provided some help from course tutors during their early learning phase. A database-manipulation language, SOLO, has been specially designed (Eisenstadt, 1978) for them and offers an easy access route to high level programming concepts. We define a novice as a person who is not conversant with other programming languages and who has read all of the SOLO programming manual and completed three 'course assignment' Study Center Activities before setting out to perform experimental tasks.

A concise summary of one of these problems (to be discussed in some detail in Section 3 below) is:

Given a database describing objects piled up on one another as follows:



write a program which simulates the effect of someone firing a very powerful pistol aimed downwards at the topmost object (SANDWICH), yielding the final database shown below:



The solution to the problem, in SOLO, involves writing only two lines of code:

```

TO SHOOT /X/
1 NOTE /X/ HAS BULLETHOLE
2 CHECK /X/ ON ?
  2A If Present: SHOOT *; EXIT
  2B If Absent: EXIT
DONE

```

We have performed detailed studies of six Subjects (and other Subjects in less detail) who were given this problem to solve. The studies include concept rating, recall, and grouping tasks, transcription tasks, program understanding tasks, questionnaires, and verbal protocols taken while they worked on the specific problem. Of those we studied, only two produced an adequate solution. There are obvious differences between expert programmers and novices, but there are also intriguing differences among novices themselves. We distinguish 'talented' from other novices, and in this paper we try to indicate a couple of ways in which these two groups differ and ways in which they are alike. But first, we place our research in the context of other recent research into problem solving behavior.

## 2) BACKGROUND AND RELATED MODELS

We are interested in providing a model of the mental processes which occur when novice programmers transform a verbal statement of a problem into a programmed solution to the problem. Problem solving in such a task involves understanding what the problem is, finding or devising an algorithm, and writing and debugging code. This is the usual concern of anyone studying problem solving, and our model is in some respects an instance of what can be fairly described as the 'modal model' of problem solving.

The modal model of problem solving exhibits the following features. Firstly, there is a phase of 'problem understanding' during which the problem statement is transformed into a mental structure which represents those aspects of the problem which are critical to its solution. Secondly, there is a phase somewhere between problem understanding and the running of 'solution' processes, that is variously described as 'method finding', 'general solution approaches', 'schema activation', and so forth. It is during this stage that domain knowledge is accessed, knowledge which acts as a bridge between understanding and solution processes. For example, a problem statement in programming may contain information about database structures and alterations to be made to those structures but not mention recursion as an appropriate method for achieving the desired results. The person solving the problem must recognize the relevance of the particular method on the basis of the given features of the problem statement. Finally, there is the 'solution' phase itself. Often this stage is seen as somewhat less problematic than the other phases. Access to particular structures in the method-finding phase is often thought to provide direct information about the way the method should be used to transform the mental representation of the problem into a solution. That is, the method is often viewed as a solution framework into which elements of the problem are slotted, although other processes may then operate on the instantiated and filled framework.

The modal model has also been used to explain problem solving in areas in which there is no 'reading' phase - like go and chess. In these problems reading is replaced by 'board scanning', understanding is equivalent to 'recognizing meaningful patterns', and method finding involves selecting the best move from the alternative possibilities. Experts are described as having the ability to classify large numbers of meaningful patterns and large configurations of meaningful patterns whereas few and simple patterns trigger the corresponding knowledge of novices.

Very often models of problem solving featuring these three phases are only really concerned with one of the phases in any detail. Below we discuss three such models - each selected because they cover a different aspect of the model and because they are related to or differ in some important respect from our own model.

### 2.1) The Understanding Phase.

A well known model of the understanding stage in problem solving is the UNDERSTAND program of Hayes & Simon (1976). In their model 'problem understanding' involves two subprocesses, Language Understanding and Model Construction. The product of these processes is a problem space containing the initial and goal states of a problem, the problem objects and their properties and relations, and, finally, the operators for transforming the initial into the goal state, plus any restrictions on the use of the operators. This final representation can then be operated on by a special purpose problem solving mechanism. If the problem is not solved in this 'Solution' phase, the problem understanding mechanisms are instantiated a second time and the process begins again.

The UNDERSTAND program operates on 'well defined' problems. These are problems that specify all the information a problem solver needs to solve the problem. A problem with this model from our point of view is that programming problems are not well defined, in the sense that 'operators' are not usually explicitly indicated in programming problem statements. Programmers have to recognize which operators are relevant from an analysis of various features of the problem statement. If the problem does not cue a known algorithm, then the programmer must devise one. Brooks (1977) makes much the same observation and introduces the notion of 'method finding' to account the programmer's need to determine a suitable algorithm, but he has nothing to say about method finding processes other than to indicate reflections of such processes in the protocols he analyzed. Part of our goal is to demonstrate such processes in operation.

Equally important, in UNDERSTAND, 'Solution Process' do not begin operating until enough is known about the problem to get the processes going. Unfortunately, this 'enough' involves knowing everything. In 'well defined' problems solution processes cannot get started until the operators (and the restrictions on their use) are known, and in the 'Tea Ceremony' problem, which is used by Hayes & Simon to exemplify the operation of the system, this information is not finally given until the penultimate line of the problem statement. Thus, problem understanding processes necessarily precede any attempts at a solution to the problem.

In programming the solution process actually begins where problem understanding begins: at the first line of a problem statement. In our model Understanding and Solution processes co-occur. Indeed, in many cases, it would be difficult to distinguish between 'understanding' a problem and knowing the solution to the problem. If, while reading the problem statement, one of our Subjects states - "Oh, this is going to be like the 'Infect' program. I just have to insert these two triples into that framework and that's it!", - and if the Subject has a working model of the behavior of the Infect program, then he has solved as well as understood the problem in the same moment. The novice may not have much knowledge of programming, but what little he has is brought to bear on understanding and solving the problem wherever opportunities arise.

## 2.2) Schema Activation.

Chi, Feltovich & Glaser (1980) have undertaken extensive investigations of the cognitive structures which novices and experts have acquired in the domain of physics, and the manner in which this knowledge is indexed when these disparate groups set out to solve physics problems. They show that the expert's schemas are organized in terms of physics solution principles, while novice's schemas are based in 'object' categories with pointers to equation formulae for specific problems.

Chi, Feltovich & Glaser show that both experts and novices proceed in solving problems by first categorizing the current problem. This categorization makes available information which can be used to guide further understanding/ solution processes. The major problem confronting the physics expert - the same problem confronting the chess expert - is in choosing amongst candidate schemas. Solution processes are more or less non-problematic once the correct schema has been selected from the candidate schemas.

Finally, Chi, Feltovich & Glaser conclude that novices are 'stimulus bound' - that they are influenced more by characteristics of the problem text (keywords) than by the principles of physics that underlie a wide range of such problems. The expert, on the other hand, is able to derive second order problem characteristics from the same textual features that influence novices and candidate schemas are then keyed by these second order representations.

Another way of talking about the novices discussed by Chi, Glaser and Glaser is to say that a novice can read through a problem statement and even develop a solution without ever discovering what the problem really is. This is the position which we take. Our view is that problem solving involves constructing and running a mental model of a problem, and that programming and world-knowledge interact to direct and constrain the mental models that are constructed. The notion will be elaborated below.

### 2.3) The Solution Phase.

A model of the running of solution processes by experts in the writing of computer programs - given an adequate representation of the problem plus an appropriate mediating algorithm - has been provided by Brooks (1977). In Brooks' model the most complex processes are those concerned with determining the effects of a piece of code, and updating the problem model once a segment of code has been generated. Code generation itself is a more or less straightforward translation process. Coding failures occur, of course, but this is because the coding rules are inadequately specified or are generated in 'circumstances for which [they are] not appropriate' (Brooks, 1977). In our studies we find that novices have little difficulty with coding per se - other than minor syntactic details, which are handled automatically by SOLO - but there are large differences in novices in their ability to evaluate a segment of code once it has been generated. A large part of our investigations have been concerned with discovering the evaluation rules used by novices once they have generated a segment of code.

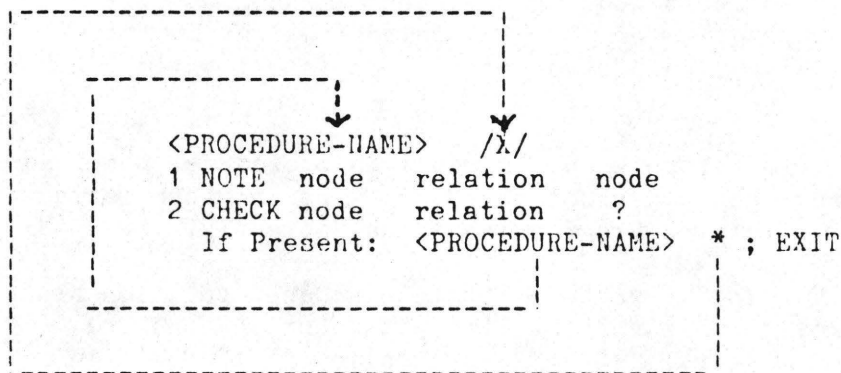
Although we are in essential agreement with the general outline of this modal model of problem solving it has a number of weaknesses. Even in simple domains like puzzles there is evidence that transfer of learning easily occurs only in particular conditions (Reed, Ernst & Banerji, 1974; Hayes & Simon, 1974; Luger & Bauer, 1978). In complex domains like mathematics the problems of transfer are magnified (Schoenfeld, 1980). Schoenfeld has shown that possession of a relevant store of knowledge is not a sufficient basis for solving mathematics problems. The development of skill in this domain involves the construction of complex indexes to that knowledge. Schoenfeld demonstrates that indexing is not always a simple matter of 'keying' knowledge but may involve applying heuristics to heuristics. That is, the novice must be taught not only useful rules of thumb for accessing relevant information, but must also be taught heuristics for selecting amongst accessed concepts. We will show below that even without 'indexing problems', that even when novice programmers are 'led by the nose' to a problem solution (a program that the Subject can 'imitate' or 'copy') problem solving is often a laborious and problematic task that ends in failure.

Novice programming behavior is only partially explainable in terms of retrieval and direct application of schemas acquired during a fairly brief training phase. As focus shifts away from experts and onto novice and naive problem solvers, explanations based on domain specific knowledge tend to be supplemented with general world knowledge. The processes through which these different sources of knowledge are said to interact are various, depending on the researcher, but they are often now discussed in terms of the construction and evaluation of mental models (Gentner, 1981; di Sessa, 1981; Eisenstadt, Laubsch & Kahney, 1981; Norman, 1982; Kahney & Eisenstadt, 1982). The general notion behind mental models is that of a cognitive structure that is constructed in working memory and 'run off' in order that its behavior might be observed. The mental model that is constructed is presumed to be a function of a large number of factors, such as the extent of conceptual knowledge, the number of concepts the person has mastered, the rules for combining and evaluating models, and so forth. We shall not discuss most of these issues in this paper. In the next section we will discuss differences between novices in terms of their mental models of SOLO programming concepts and in the final section of the paper we will briefly discuss an interpretation theory for

scoring verbal protocols. The theory conforms, in general outline, to the modul model described above.

### 3) MENTAL MODELS

To reiterate something that has already been said, there are obvious differences between expert programmers and novices, but an equally important distinction is that between 'talented' and 'average' novices. That is, given the same material to study, and an unlimited amount of time in which to master the material, some newcomers to programming 'get it' and some don't. Talented novices differ from the average in the number of programming concepts with which they are familiar and in the degree to which familiar concepts have been understood (i.e., two novices may both be familiar with the NOTE and PRINT primitives in SOLO, but one may think they both serve the same function). Talented novices have much in common with experts in terms of the way their knowledge is organized, and in the way in which it is brought to bear in solving programming problems (Kahney, 1982). Unlike average novices, much of the new knowledge of the talented novice is organized in memory as 'plans' for achieving particular program-effects, such as 'Conditional-side-effect-on-a-database', or 'Generate-next-object-and-side-effect-each' (Eisenstadt, Laubsch & Kahney, 1981). We have no evidence that 'talented' novices spend more time or effort in understanding programming; we have evidence that 'average' novices spend considerable time trying to come to grips with concepts like recursion. But the talented novice is one who develops a model of 'the way recursion works' (albeit, an inaccurate model in many respects) while the average novice commits a segment of code to memory with the rule that the segment has a particular effect without having a model of the way the effect is achieved. The model which talented novices develop of recursion can be graphically described as:



That is, recursion is seen as a kind of looping operation where successive nodes in a particular database structure are 'fed back' to the 'top' of the procedure. The effect causes the program to start up again and perform whatever action is indicated at Line 1 of the program on the node delivered to the parameter slot. We have derived this model from one Subject's performances on various tasks, during one of which the Subject can be seen on film indicating the loop with her pencil. The task involved understanding a program written by another novice. When reading this segment of code:

```

CHECK /X/ WORKSFOR ?
If Present: IMPLICATE * ; EXIT

```

the Subject says: "Check X works for.... somebody. If so...." [at which point the Subject used her pencil to trace a loop from the word IMPLICATE back up to the title line of the program] "...back to the beginning." S8 uses the coding framework for writing her own recursion programs and also as a model which can

be manipulated to determine its output. S8 produced the 'correct solution' program that is provided on page 1 of this paper and the following is a summary of the protocol produced by this Subject after the program had been typed in at the computer terminal but before it was run:

Right, so.... To Shootup X, let's say X is a sandwich.... It notes in the database X has bullethole. It then checks whether X is 'on' anything. X is on plate, so it will do that to plate. So that should keep on doing that: plate's 'on', check, and so on and so on. If it's not 'on' anything it's okay to just exit. Right.

Up to this point S8's discussion and coding of the program has been quite abstract in the sense that only programming constructs are used or discussed. But when the program is evaluated, variables are replaced by database objects like 'sandwich' and 'plate' and the recursive segment evaluated to the depth of the second node ("X is on plate, so it will do that to plate"). The next sentence indicates something like "If it'll do it to the first two nodes, it'll do it to all of them."

The average novice on the other hand writes recursion procedures without a model of the behavior of the program, and therefore needs to use the computer to evaluate code. S5, who is not untypical of the average novice in this approach to writing programs, made several attempts to produce a procedure that would produce the required output. Since this Subject had no model of the behavior of the programs written on successive attempts, all the Subject could do was type a program in at the terminal and 'wait for the computer to evaluate it'. If the desired output occurred the program worked (and one of our novices did write a program which produced the desired output - for the wrong reasons - but could not say how the program achieved its effect) and if not, the program had to be debugged. The usual strategy adopted by novices without models of program behavior is to 'imitate' a program found in the programming manual. These programs are not hard to find as the first line of all our problem statements indicate where to look in the Programming Manual if help is needed with the design of a program for the particular problem. Summarized extracts from the protocol of S5, each taken after a different attempt to write a program, are these:

Experimenter: What's going to happen when you run this procedure? S5: I hope it will go through that sequence and shoot the floor. The database is in and I've copied that program [from the Programming Manual] exactly.

This example in the book.... You've got one 'state' and one 'relationship' and in my example we've got three things. We've got two states and one relationship. And that's not fair. I thought that following the example in the book would lead me through automatically, but I've decided the problem's got more, um, states in it than the problem in the Unit.

Will it accept.... Well, I can always type it in and try.... And it will correct me won't it?

Meaningless example in the text.

The keywords and phrases in this protocol are 'hope', 'copied', 'following this example' and 'I can always try it and see'. Trial and error, the last hope of a person with no other resources. Even so, this short extract is insufficient to indicate the grief experienced by many novices who find this problem beyond solution.

### 3.1) A Program Transcription Task.

Differences between talented and average novices are found in the entire range of tasks they are asked to perform. Program transcription is a task in

which programmers are simply required to copy a program which they are allowed to view five times for a period of ten seconds on each viewing. All Subjects are told that they should try to transcribe the program in as few attempts as necessary. Extraction of information is stressed; they are told to guess wherever possible, as they are allowed to cross out errors that are discovered on subsequent viewings.

The 'program language' used in the task is an abstract version of SOLO. Although neither the novices or experts had ever seen the 'language' before performing the task it is presumed that the novices have an advantage in that the relationship between SOLO and its abstraction, while the experts have the advantage of knowing there is a structure to be extracted, and this structure is an extremely simple one. The subjects are given five different colored pens and use a different color for each recall attempt. We are thus able to determine the amount and nature of the information extracted on each viewing. Figure 1 is a copy of the program we use for analysis. It is the last of three programs which Subjects are required to transcribe, so all subjects by this time have had practice in transcription and opportunity to learn the structural features of the language.

```
TEST flat has 2 bedrooms
  IF YES: looks promising ; CONTINUE
  IF NO: too bad ; EXIT
TEST rent less than £100
  IF YES: really looking good ; CONTINUE
  IF NO: too expensive ; EXIT
TEST neighbors are friendly
  IF YES: keep looking ; EXIT
  IF NO: take it ; EXIT
```

Figure 1.

In Figures 2, 3, 4 & 5, the 'framework' is given of the first recall of the expert, the talented novice, S8, and two average novices, S5 & S10, respectively. The expert currently writes programs in Assembler (MACRO-20) and Pascal and has considerable experience in writing FORTRAN and BASIC programs, and some experience in ALGOL 68.

```
TEST flat has 2 bedrooms
  IF YES:          ; EXIT
  IF NO: CONTINUE
TEST
  IF YES:          ; EXIT
  IF NO: CONTINUE
TEST
  IF YES:          ; EXIT
  IF NO: CONTINUE ; EXIT
```

Figure 2.

```
TEST flat has two bedrooms
  IF YES: looks promising; CONTINUE
  IF NO: too bad ; EXIT
TEST
  IF YES:
  IF NO:
TEST
  IF YES:
  IF NO:
```

Figure 3.

```
TEST flat has 2 bedrooms
  IF YES: looks good ; CONTINUE
  IF NO: too bad
```

Figure 4.

```
TEST flat has 2 bedrooms
  IF YES: looks promising : CONTINUE
  IF NO: too bad : EXIT
TEST
```

Figure 5.

As can be seen (Figures 4 & 5), even after considerable practice at transcription, the typical novice is still extracting information from the programs as one would extract information from a novel: a line at a time. S5 needed all five allowable viewings to transcribe the program perfectly. The strategy used by S5 was to extract information from each 'TEST' block at a time, and to

search for errors in the immediately previous recall attempt. S10 had a correct transcription after four viewings of the target program, and the recall strategy was essentially the same as that of S5.

Both S8 and the Expert (Figures 2 & 3) however extract the syntactical structure of the program plus the first few lines of program text on the first viewing. The expert extracted less text and more of the structure than S8 on this first viewing. The expert wrote and crossed out CONTINUE on the last line of his recall and replaced it with EXIT - presumably using his knowledge of programming to correct his recall. On the second recall the expert corrected the flow of control errors (he'd got them backwards on the first recall - an error he might have avoided then if he had extracted any of the textual information) and recalled four lines of text. Altogether the expert took four viewings before transcribing the entire program.

S8 also required four viewings to transcribe the entire program. On the second recall S8 extracted the rest of the text, although the last two lines of text were recalled in the wrong order. This was corrected on the third pass, and the final line of text was added on the fourth. The point is that the talented novice has extracted a model of the important characteristics of program structure, and the model is not unlike that of the Expert.

#### 4) AN INTERPRETATION THEORY FOR PROTOCOL ANALYSIS

In this section we briefly discuss the behavior of novices from the point of view of the strategies they adopt when confronted with a problem statement. We argue that novices and experts are alike in bringing whatever programming knowledge they have to bear as early as possible in the reading of a problem statement. This is the same as to say that solution processes operate in parallel with understanding processes. Skilled novices have an advantage over unskilled novices in that they have program models, as discussed above, which can be brought into the service of understanding mechanisms. Thus, if subject S5 has a model of 'recursion' and this model is triggered early in the reading of the problem, then the model can be used to direct understanding processes to important aspects of text. We hypothesize that the programming problem solving behavior of a novice can be characterized in terms of the following types of behavior:

- 1) Method finding. The novice will attempt to classify the problem in terms of problem types with which he is already familiar. This behavior is related, of course, to reading a line or several lines from the problem statement.
- 2) Evaluation of candidate methods. This will involve a search for information which would instantiate a candidate method. A particular method might require a particular type of database structure, for example. If so, we would expect to see the problem solver actively search for such information if the method is selected as a candidate.
- 3) Instantiation of a method. The novice will begin thinking in terms of the way the problem can be realized in a program. When a method is instantiated the novice will assimilate what is already known about the problem to the instantiated schema, and will use the schema for problem solving - setting up expectations about what is to come, to detect conflicts between what is known and what is expected, and so forth.
- 4) Coding. This is self explanatory.
- 5) Evaluation of code. The novice will run a mental model of the behavior of his program. If evaluation indicates the code is successful the novice will go on to (6) below. Otherwise he will go through the previous steps in descending order (4,3,2,1).
- 6) Testing. The novice will type his program in at the terminal and run it.

We have formulated the theory as a set of rules for scoring the verbal protocols provided by our Subjects while they were writing programs. The



smallest segment of protocol used in our analyses is a sentence, or a clause if there is a prominent pause in the Subject's speech. Each line of protocol is assigned to one of various behavior categories. A simple example of a rule for scoring the protocols is in assigning a line of protocol to the READING category. A line of protocol is scored as an instance of READING when the Subject reads a line from the problem statement.

The program writing behavior of novices is not random, but guided by various strategies. A strategy is a patterned sequence of behaviors all of which are related to a certain goal. We need to be able to subsume larger segments of protocol than the single line if we claim that we have a 'theory' of novice programming behavior, and in support of such a claim we examine the 'derived' strategies of our Subjects by applying the interpretation theory to the protocols and comparing the strategies with the theory outlined above.

Strategies are 'derived' by applying the scoring rules to each segment of protocol. In the remainder of this paper we shall be concerned only with one type of segmentation, involving the Reading phase. The Reading phase begins when the Subject reads the first line of the problem statement and ends when he has read all the problem statement. The Subjects discussed below were asked to read the problem statement one line at a time and to tell the experimenter everything they knew at that point in time - what the problem was, how they might tackle it, any predictions they had - before going on to the next line of text. They could make notes, consult notes or the programming manual, or any other source of information, as and when they liked. The analysis below concerns the scoring of the first segment of Reading protocol - everything said between reading the first and second lines of the problem statement - from Subjects S8 & S5.

In summary, we hypothesize that the primary strategy employed in program writing is finding a classification for the current problem, and that attempts at classification will occur early rather than late in problem understanding. When a problem can be related to a class of problems for which solution techniques are already known, a major part of problem solving has been circumvented. The 'Classification' strategy involves RETRIEVAL of a candidate schema, REHEARSING the schema, EVALUATING it with respect to the current problem. If the result of the EVALUATION is positive then the schema will be INSTANTIATED, which leads to PREDICTING the content of the rest of the problem statement, or ASSIMILATING what has already been read to the instantiated schema. ASSIMILATION cannot occur after only the first line of the problem statement, since the first lines of our problem statements never contain anything that might be assimilated to a schema. A strategy is initiated by READING of a line from the problem statement. Figure 5 shows the category configuration of the Classification strategy.

CLASSIFICATION:

- (A) READING-1
- (B) RETRIEVAL
- (C) REHEARSAL
- (D) EVALUATION
- (E) INSTANTIATION
- (F) PREDICTING or ASSIMILATING

Figure 5.

Here are the protocol scoring rules for interpreting lines of the protocol (Instantiation and assimilating are not discussed, as the protocols do not contain instances of these categories of behavior):

- 1) READING: A person reads a line from the problem text.

- 2) REREADING: A person rereads a line from the problem text, except the last line. That is, if a person has read lines 1, 2 & 3 and rereads either/both line 1 or 2, then REREADING has occurred.
- 3) FOCUSSING: A person rereads the line of text last read. That is, if a person has read lines 1, 2 & 3 and rereads line 3, then FOCUSSING has occurred. The behavior is assumed to be related to RETRIEVAL failure.
- 4) EXPERIMENTER-COMMENT: Any comment made by the experimenter. The experimenter is indicated by 'E:' at the front of a line of protocol.
- 5) RETRIEVAL: We consider that retrieval is equivalent to activating some structure in LTM. Activation of a structure is sometimes signalled by Subject statements such as "That reminds me of iteration". This statement signals the activation of the 'iteration' schema. RETRIEVAL is often a response to READING, REREADING, or FOCUSSING, and a line of protocol immediately subsequent to these processes should be examined for indications of this process.  
  
RETRIEVAL may either be 'cued' or 'associative'. We regard responses to READING, REREADING, and FOCUSSING as cued RETRIEVAL. The concepts that are used subsequently to cued RETRIEVAL of a particular concept are considered to be associatively related to the particular concept, and made available through an associative link. If a person READS "On page 80 of [the Programming Manual] we looked at a method for making a particular inference 'keep on happening'", and responds: "Is that 'iteration'?" then 'iteration' is an example of cued, or C-RETRIEVAL. If the Subject then says, "Or maybe it's 'recursion'" then 'recursion' is an example of associative, or A-RETRIEVAL.
- 6) PROBE: A person PROBES what he knows about a concept by explaining it to himself (aloud) or to the experimenter, or when he provides a definition or makes an attempt to construct a meaning for a particular concept, and so forth. An example would be if a person thought the current problem had something to do with recursion (a RETRIEVAL) step, but then had to determine what 'recursion' might be (e.g., "Is that where you use.... etc."). We know from pilot studies that both talented and average novices 'probe' their knowledge of a particular concept before trying to apply what they know.
- 7) EVALUATION: This activity takes several forms. EVALUATION occurs when a person determines the output for a segment of code he has written. Also, EVALUATION occurs when a person attempts to determine the appropriateness of a particular algorithm for the current problem. It may sometimes be difficult to distinguish EVALUATION from REHEARSAL.
- 8) PREDICTION: The person states an expectation of some sort, generally about the type of program that is required, or about information that is yet to be given, and so on.
- 9) META-COMMENT: The person explains why he thinks some thing, or comments on the strategy being followed or in some way indicates the processes that are occurring in his quest for a solution to or understanding of the problem.

The first protocol below is that of S8. At the left of each line of protocol (the different lines are indicated by numbers in brackets) there is the category label which has been applied to that line. Lines 2, 5, 8 and 10 have been deleted from this protocol as have several lines from the protocol of S5. The deleted lines had been labelled 'UNACCOUNTED' as they could not be classified in terms of the categories indicated in the interpretation theory. Invariably these lines contained single words such as "Um...." or "Well....". No account of this type of comment will be given in this paper.

(PROTOCOL, SEGMENT-1, S8)

- READING (1) "On page 80 of Units three to four we looked at a method for making a particular inference 'keep on happening'."
- C-RETRIEVAL (3) Is that called 'iteration'?
- A-RETRIEVAL (4) No, 'recursion'.
- PROBING (6) I can remember something about being told something about the distinction between iteration and recursion and one goes sort of like along a database and the other sort going down.
- META-COMMENT (7) Well, that's how I sort of thought of it.
- EXPERIMENTER-COMMENT (9) E: Alright. Any further expectations or any....?
- EXPERIMENTER-COMMENT (11) E: I just want to know everything you, uh.... I mean as you read it....
- PREDICTION (12) So.... I think this is going to say something about what happens when you keep on applying a

Figure 6 summarizes the derived strategy for this protocol segment.

S8:

READING  
RETRIEVAL  
PROBING  
PREDICTION

Figure 6.

The structure of this first segment of S8's protocol is in good agreement with the theory. The reading of the line results in the retrieval of relevant information at a fairly high level of abstraction (recursion and iteration). And the retrieval itself is followed by self probing. The exception is the last category, PREDICTION. We associate this category of behavior with an 'Instantiation' strategy (not further discussed here) and we are led to infer that at this point S5 has already instantiated one of the candidate models mentioned in lines 3 & 4. Subsequent lines in the protocol confirm this inference (further details in Kahney & Eisenstadt, 1982).

(PROTOCOL, SEGMENT-1, S5)

- READING-1 (1) "On page eighty of Units three to four we looked at a method for making a particular inference 'keep on happening'."
- FOCUSSING (2) Keep on happening....
- META-COMMENT (4) I am very surprised, because I didn't think it was going to be detailed like this.
- FOCUSSING (6) An inference keep on happening....
- FOCUSSING (9) Keep on happening....

- META-COMMENT (11) That reminds me of something.
- META-COMMENT (12) I'm trying to remember what the words were that it reminds me of.
- FOCUSSING (14) Keep on happening....
- C-RETRIEVAL (16) For-Each-Case-Of, it reminds me of.
- C-RETRIEVAL (17) And then 'inference' is where you have, um....
- PROBING (18) Two statements, and the database is able to form, um....
- PROBING (19) Or the program is able to modify the database by, um....
- PROBING (20) Putting in a new relationship on the basis of relating to relationships that were already in the database.
- PROBING (21) So you could have two triples and they would imply, or the....
- PROBING (22) The program has been made so that, um....
- PROBING (23) The existing two triples will form an inference which is formed....
- PROBING (24) Which is stated on....
- PROBING (25) ...the database as a third triple.
- PROBING (26) That's what an inference is.
- FOCUSSING (27) We looked at a method for making [an]... inference....
- META-COMMENT (28) I can't remember what it is.

Figure 7 summarizes the derived strategy for S5.

S5:

READING  
FOCUSSING  
META-COMMENT  
FOCUSSING\*  
META-COMMENT\*  
FOCUSSING  
C-RETRIEVAL\*  
PROBING\*  
FOCUSSING  
META-COMMENT

Figure 7.

The structure of this segment of S5's protocol also is in good agreement with the theory. Focussing, meta-commenting, retrieval, and probing are indicated to occur on subsequent lines of protocol by an asterisk following the category

names in the derived model of strategies. In this segment the Subject reads and focusses as an aid to retrieval, and continually probes what is retrieved.

The major difference between S5 and S8 in this segment is that S8 has an internalized model of recursion which has been indexed (presumably by a key phrase, such as 'a method for making an inference keep on happening) and instantiated, while S5 is left 'focussing'. The differences throughout the remainder of the two protocols are much more pronounced. S8's protocol indicates predictive understanding and a workable solution before the problem statement has been entirely read. S5 never really succeeds in understanding what the problem is, and all efforts at 'imitating' the solution indicated in the first sentence of the problem statement come to grief.

There are also counterexamples to the theory. Here is an extract from the first segment of protocol of S11:

(PROTOCOL, SEGMENT-1, S11)

- READING (1) "On page 80 of Units three to four...."
- META-COMMENT (2) And then I looked at that again because I immediately tried to think of what page eighty might be like, and Units three to four.
- META-COMMENT (3) I couldn't remember.
- READING (4) "...we looked at a method for making a particular inference 'keep on happening.'"
- META-COMMENT (6) As I was reading that I started to think....
- META-COMMENT (7) About what I had read....
- META-COMMENT (8) And it rang a few bells, but....
- META-COMMENT (9) I wasn't very clear about it.
- META-COMMENT (11) The procedure you need for that, so I decide to carry on reading.

Here is the structure of the derived strategy:

S11:

READING  
META-COMMENT\*  
READING  
META-COMMENT\*

Essentially, this Subject is reporting that some of the words and phrases are recognizable - method, inference, 'keep on happening' - but the corresponding structures in memory are not activated. Each segment of the reading protocol for this Subject looks much the same as the segment provided here.

## 5) CONCLUSION

We believe that there is a useful distinction between two types of novice - those we have called talented, and the rest. We have studied 6 novices in considerable detail, and find that a talented novice is like an expert in many respects in which the average novice is quite unlike the expert. A good

example is the behavior of novices on the transcription task by comparison with an expert. These differences hold across a variety of task situations that we have studied, most of which have not been discussed in this paper.

There is a great deal more to be said about mental models, about the theory of novice problem solving, about the interpretation theory, and about the various experimental methods used in studying novice programming behavior, than has been indicated in this brief paper. We hoped simply to indicate the direction our research has taken and a very brief of some of our findings. We should have liked to present more detail about each of these matters, for the final picture is far from simple. For example, a talented novice may not only find his mental models an aid to understanding, but a hindrance as well. S8, who believed that there were only two types of program you could write in SOLO - iteration and recursion - began working on the second problem thinking it would be a problem in iteration because a problem in recursion had previously been given. The preconception was mistaken, and the Subject's 'iteration model' led the Subject astray for a considerable length of time in solving the second problem given. Mental models are more fully discussed in Kahney & Eisenstadt (1982). A detailed discussion of all the issues raised here can be found in Kahney (1982).

#### REFERENCES

- Brooks, R. Towards a theory of the cognitive processes in computer programming. *Int. J. Man-Machine Studies*, 9, 1977.
- Chi, M.T.H., Feltovich, P.J. & Glaser, R. Categorization and Representation of Physics Problems by Experts and Novices. *Cognitive Science*, vol. 5, 1981.
- di Sessa, A. A. The role of experience in models of the physical world. *Proceedings of the Third Annual Cognitive Science Society Conference*, Berkeley, California, 1981.
- Eisenstadt, M. Artificial Intelligence Project. Units 3/4 of 'Cognitive Psychology: a third level course.' Milton Keynes: Open University Press, 1978.
- Eisenstadt, M., Laubsch, J. & Kahney, H. Creating pleasant programming environments for cognitive science students. *Proceedings of the Third Annual Cognitive Science Society Conference*, Berkeley, California, 1981.
- Gentner, D. Generative Analogies of Mental Models. *Proceedings of the Third Annual Cognitive Science Society Conference*, Berkeley, California, 1981.
- Hayes, J.R. & Simon, H. Understanding written problem instructions. In Gregg, L.W. (Ed.), *Knowledge and Cognition*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1974.
- Kahney, H. & Eisenstadt, M. Programmers' mental models of their programming tasks: the interaction of real-world knowledge and programming knowledge. *Proceedings of the Fourth Annual Cognitive Science Society Conference*, Ann Arbor, Michigan, 1982 (in press).
- Kahney, H. An in-depth study of the behavior of novice programmers. Technical Report No. 82-9, Human Cognition Research Group, The Open University, Milton Keynes, England, 1982.
- Luger, G. & Bauer, M. Transfer effects in isomorphic problem situations. *Acta Psychologica*, 1978.

Norman, D. Some observations on Mental Models. CHIP Technical Report No. 112, Center for Human Information Processing, University of California, San Diego, California, 1982.

Reed, S.K., Ernst, G.W. & Banerji, R.B. The role of analogy in transfer between similar problem states. *Cognitive Psychology*, vol. 6, 1974.

Schoenfeld, A. H. Can Heuristics be Taught? In Lockhead, J. & Clement, J. (Eds.), *Cognitive Process Instruction; Research on Teaching Thinking Skills*. The Franklin Institute Press, 1980.





Measuring the performance of students in an  
introductory informatics course

R.P. van de Riet  
Computer Science Department  
Vrije Universiteit, Amsterdam

This paper is based on the article: Performance  
Evaluation of the BASIS system by R.P. van de Riet,  
published in the Proceedings of the 4th International  
Symposium on Modelling and Performance Evaluation  
of Computer Systems, Vienna, Austria, February 6-8  
1979; North Holland Publishing Company.

ABSTRACT

BASIS is an interactive system, based on PASCAL, for the workshop of the introductory course in informatics. It has built-in facilities for evaluating itself and the performance of the students. The aim of the performance evaluation is to have a tool by means of which the system and the course can be gradually improved.

1. INTRODUCTION

The BASIS system is used in the introductory course in informatics as the primary tool for the student's practical work. It is an interactive system for both program composition and program testing. The language is a subset of PASCAL [1] (no records, no sets, no subranges, no pointers, only one data file and no goto's). The only way a student can make a program is by making procedures which can be individually tested. In fact, for the student a program is just the collection of variables and procedures he introduced. The emphasis is on structured programming with short, well documented, procedures. The current BASIS version checks if the procedure text conforms to a simple but adequate lay-out structure and also whether the text contains any form of comment. The editor is a large subset of the UNIX editor [10].

In two preceding IFIP conferences we reported about the design criteria and plans [5] and about the implementation of the system [6]. In this paper we want to discuss several measurements which have been carried out. These measurements concern primarily the functioning of the system in response to the student and vice versa. A major objective which we want to realize with the system is that of more or less automatic upgrading. Not in the sense that bugs are removed (actually the current version of the system is very stable), but in the sense that reactions of the system to student behaviour are improved.

There are several ways to measure the system-student responses in order to make improvement possible. One method is to question the students about the system by means of questionnaires. In an early stage of the development of the system this has been carried out by two psychology students on a group of alpha students (from the humanities). Very few problems were signalled in this way which were not already known by personal communication. In particular, the placement of the semicolon and the use of the editor turned out to be troublesome. This way of measuring the system was not pursued any longer; although it is not impossible to redo such an investigation in the future if some psychologists show interest. Another method, which will be extensively reported in this paper, is to automatically analyze the conversation between system and student. In this way it turns out to be possible to get a clear picture of what an average student does, how he reacts upon errors, which errors he makes, which constructs he uses, etc. and of the behaviour of the system in terms of response time, error messages (whether they are clumsy or not), etc.

The structure of this paper is as follows. In section 2, we will demonstrate a typical session of a student. In section 3, we will show how a so-called stat-file is constructed from the system-student conversation. In section 4, we analyze the global

behaviour of the student and we compare several types of students: informatics and mathematics, biology and geology, and students from the humanities. In section 5, a detailed analysis of the errors will be given in terms of reaction time, think time, frequency, adequacy of help information and repetition of errors. Several correlation coefficients will be given also for the different groups mentioned above. In section 6, we will outline how the errors are distributed in time. In section 7, we will report about the analysis which has been performed for each individual error and how this influenced a new version of the system. In section 8, the use of the language constructs will be described. This analysis reflects a little bit the analysis of Knuth [4] about use of FORTRAN constructs and the analysis of Tanenbaum [9] concerning the use of PASCAL constructs. These investigations were designed for optimization purposes of compilers and underlying machines.

Our goal is to know what the student is doing from a pedagogical standpoint so that the course can be improved pedagogically. In this sense our investigations are in the same area as the studies of Sime and Guest [8] where they measure the use of certain language constructs as e.g. if-then-else versus goto's, or the investigations of Gannon [2] who describes several controlled experiments where programmers use (more or less) structured programming tools. In section 9, we report the results of the measurements concerning the system responses, together with a short overview of past measurements by M. Kersten [3] concerning the internal functioning of the system components. Here, we also give some numbers concerning size and speed of the system and the hardware configuration. Finally, in section 10 we describe some future plans.

## 2. AN EXAMPLE OF A BASIS SESSION

We suppose that the student is somewhere in the middle of the course so that he is already familiar with the notion of variables, types, values, procedures, editing, etc. He will work on a problem where the main procedures have been thought out and written at home. (In fact the course assistants take care that the students do their homework at home and not behind the terminal). The problem is to calculate the n-th Fibonacci number  $f[n]$ , defined as  $f[0] = 0$ ,  $f[1] = 1$ ,  $f[i] = f[i-1] + f[i-2]$ , for  $i > 1$ . This problem is identified as exercisel. A possible interaction is shown below. BASIS normally ends its response with an arrow "-->" after which the student gives a next command. If the student types in a procedure (or function), then the BASIS reaction upon a new line is "...", so the student can easily see if he is still typing in the procedure or that he has finished the procedure. In general, BASIS responds with "." if the command is not finished.

When the student is editing a procedure, with the name "proc", by means of the command "edit(proc)", BASIS responds with "..>", if a new edit command is expected; if the edit command is not ready, as in the case of a(ppend). BASIS responds with "...".

If BASIS detects an error (syntactical or run-time) it responds

with showing the line last treated (which can be the command typed in or a line of a procedure) underlining the symbol last treated. Only if the student types in "help" will BASIS respond with "\*\*\*\*" followed by an error message. It is possible that the student asks for more help by typing in "help" again. In that case BASIS responds with some global information about the error such as a reference to the manual or some examples. Note that by explicitly asking the student for "help" instead of automatically providing him or her with the error message, we have given the system a tool to measure the student's reaction upon errors. It is now possible to measure the effect of an error message on the number of times the same error is repeated immediately, or on the time a student needs before he repairs the error (i.e. think time). Furthermore, it saves time for the student (and paper of the terminal) since in quite a number of cases (in fact 70-80 %) he apparently did not ask for the error information. In BASIS, comment is placed between "(" and ")" and may be placed in commands or in the text of a procedure. We will use it to explain what the student is doing. Let us return now to the problem mentioned above. The student starts as follows after logging in under UNIX [10].

```

*** Basis-version 230178, see: 'Basis-handleiding dec 1977'
--> var f0,f1:integer;
... type far=array[1..n]of integer;
   type far=array[1..n]of integer;
   =
--> help
*** arraybound must be an integral number
--> type far=array[1..10]of integer;
... const n=10;
... f:far;
   f:far;
   =
--> help
*** "=" expected
--> help
*** ex. const month='augustus'
*** ex. const min=10; max=20;
***           maxint=32767
*** see manual 4.1 ( const-declaration )
--> var f:far
... f0:=0;f1:=1;f[2]:=f0+f1;f[3]:=f[2]+f1;write(f[3]);
   2
--> f[4]:=f[3]+f[2];f[5]:=f[4]+f[3];write(f[5]);
   5
--> procedure fibon(n:integer);
... var i:integer;
... begin
...     while i <= n do
...         begin f[i]:=f[i-1]+f[i-2] end;
...     write(f[n])
... end
--> fibon(5);
*** remember: layout not correct, see: additions to the manual
*** remember: pr- or fu-declaration without comment

```

```

*** in: fibon
4      while i<=n do
      ==
n      = 5
i      = <undef>

--> edit(fibon)
..> l
    procedure fibon(n:integer);
..> a
... (*computes and prints n-th fibonacci number*)
... .
..> /begin f/
    begin f[i]:= f[i-1]+f[i-2] end;
..> s/end;/;i:=i+1/p
    begin f[i]:= f[i-1]+f[i-2]; i:= i+1
..> a
...     end;
... .
..> /write/
    write(f[n])
..> s/w/      w/p
    write(f[n])
..> l,$p
    procedure fibon(n:integer);
    (*computes and prints n-th fibonacci number*)
    var i:integer;
    begin
        while i<=n do
            begin f[i]:= f[i-1]+f[i-2]; i:= i+1
            end;
        write(f[n])
    end
..> q
--> (*We are out of the editor now*)
--> fibon(5);
*** in: fibon
5      while i<=n do
      ==
n      = 5
i      = <undef>

--> help
*** variable did not get a value
--> edit(fibon)
..> /begin/
    begin
..> s/n/n i:=2; f[0]:= f0;f[1]:=f1;/p
    begin i:= 2; f[0]:= f0; f[1]:= f1;
..> q
--> fibon(5)
*** in: fibon
4      begin i:= 2; f[0]:= f0; f[1]:= f1;
      =
n      = 5

```

```

i          = 2

--> list(type)
type far = array [1..10] of integer;

--> type far = array[0..10] of integer;
type far = array[0..10] of integer;
,===

--> help
*** new identifier expected
--> erase(far)
--> type far = array[0..10]of integer;
... var f:far;
... fibon(5)
      5
--> fibon(7)
      13
--> fibon(10)
      55
--> fibon(15)
*** in: fibon
  6      begin f[i]:= f[i-1]+f[i-2]; i:= i+1
          =
n          = 15
i          = 11

```

```

--> help
*** index of array bound too large
Here we skip over the session until the student has made a procedure which produces the fibonacci numbers as real numbers instead of integers.

```

```

... fibon(10)
      55.0
--> fibon(20)
      6765.0
--> fibon(40)
      1.0233e8
--> fibon(100)
      3.5423e20
--> (*this computation needed 130 real seconds*)
--> save;
--> stop
      cptime: 28.42 sec.
*** end of basissession

```

### 3. THE STAT-FILE

From the conversation, as shown in the preceding section, a file is constructed provided with some extra information consisting of real time and cpu-time used, which is called stat-file and which is used for gathering the statistics. Obviously, it is not necessary to put on the stat-file most of the system responses as "-->", full error messages and results of computations. Instead, only the error numbers are shown. Furthermore, in order to simplify the analysis of the stat- file the beginning of a procedure

declaration is signalled by a "p" and the beginning of an edit session by an "e". The two numbers with which most of the lines start are real time, measured in seconds and cpu-time measured in 20 milliseconds. These numbers are produced at the moment that the line on which they occur is sent from the BASIS system to UNIX to be put on the file. For an input line this is the moment that all the characters are put in a buffer just prior to processing the line. For an output line it is the moment that all processing is done and the line is shown on the terminal. The stat-file of the preceding section has the following form.

```

03966 26029 login
00032 00046 var f0,f1:integer;
00072 00049 type far=array[1..n]of integer;
#2103
00072 00050
00076 00051 help
00103 00054 type far=array[1..10]of integer;
00123 00059 const n=10;
00214 00060 f:far;
#2003
00214 00060
00218 00061 help
00225 00066 help
00276 00070 var f:far
00347 00071 f0:=0;f1:=1;f[2]:=f0+f1;f[3]:=f[2]+f1;write(f[3]);
00398 00078 f[4]:=f[3]+f[2];f[5]:=f[4]+f[3];write(f[5]);
00418 00086 procedure fibon(n:integer);
P
00427 00090 var i:integer;
00432 00092 begin
00445 00093     while i <= n do
00490 00095         begin f[i]:=f[i-1]+f[i-2] end;
00499 00099 write(f[n])
00503 00101 end
!
00518 00101 fibon(5);
#4204
00518 00111
00554 00114 edit(fibon)
e
00558 00116 l
00563 00117 a
00595 00117 (*computes and prints n-th fibonacci number*)
00601 00121 .
00612 00121 /begin f/
00630 00130 s/end;/; i:= i+1/p
00640 00144 a
00652 00144     end;
00653 00145 .
00660 00145 /write/
00695 00149 s/w/     w/p
00707 00154 l,$p
00723 00164 q
!
00750 00165 (*We are out of the editor now*)
00764 00168 fibon(5);

```

```

#4204
00764 00174
00777 00179 help
00793 00190 edit(fibon)
e
00799 00193 /begin/
00818 00200 s/n/n i:=2; f[0]:= f0;f[1]:=f1;/p
00824 00204 q
!
01098 00298 fibon(5)
#4002
01098 00305
01127 00310 list(type)
01159 00313 type far = array[0..10] of integer;
#2107
01159 00314
01163 00314 help
01190 00320 erase(far)
01209 00321 type far = array[0..10]of integer;
01216 00323 var f:far;
01327 00364 fibon(5)
01339 00385 fibon(7)
01357 00408 fibon(10)
01365 00440 fibon(15)
#4003
01371 00466
01379 00472 help
02766 00964 fibon(10)
02775 00987 fibon(20)
02783 01038 fibon(40)
02803 01143 fibon(100)
02971 01400 (*this computation needed 130 real seconds*)
02980 01403 save;
02986 01418 stop
=

```

Next follows a list of the keywords and their frequencies as used by the student in procedures. Note that the reproduction of the session here is not complete.

```

02986 01419 integer2
02986 01419 end2
02986 01420 begin2
02986 01420 var1
02986 01420 while1
02986 01421 dol
!

```

#### 4. GLOBAL BEHAVIOUR OF THE STUDENTS

For three courses, one for biology students (biol), one for mathematics students (math) (which includes informatics students) and one for humanities students (human), with 22, 51 and 12 participants, respectively, we analyzed the stat-files from which the following global conclusions can be drawn as depicted in table 1. The table gives numbers for the average student of the three different groups.



	Biol	Math	Human
- Total time connected with BASIS (in hours)	31	22.5	12
- Time spent while editing (in %)	24	28	13
- Time spent while typing procedures (in %)	11	17	13
- Time spent while typing commands (in %) (this includes procedure calls)	33	30	29
- Time spent while thinking about an error (in %)	23	21	3
- Time spent for chatting, coffee drinking etc. (in %)	9	4	42
- Usage of cpu (in hours)	0.74	0.58	0.06
- Number of commands including procedure calls	1404	994	476
- Number of edit calls	186	165	46
- Average number of lines per edit call	9	8	7
- Number of procedures declared	64	36	40
- Average number of lines per procedure	7	12	6
- Number of errors made	447	290	178
- Average time needed before reacting after an error was made (in sec)	59	58	66
- Number of times the same error was immediate- ly made again without asking for help (in %)	28	25	25
- Number of times "help" was called after an error was made (in %)	23	20	29
- Average time that elapsed after an error was made and before "help" was called (in sec)	36	14	28
- Number of times the same error was immediately made again after asking for "help"	18	12	18
- Number of times "help" was called two times after an error was made (in %)	3	6	5
- Average time that elapsed after an error was made and before "help" was called for the second time (in sec)	107	153	124
- Number of times the same error was immediately repeated after asking two times for "help" (in %)	16	16	19

table 1. Global behaviour of students

One can make the following conclusions from the table 1 above:

1. Mathematics students seem to work most intensively.
2. There is almost no time spent for problem solving; the time is spent for program development, as it should be.
3. Asking one time for "help" seems to help really as the number of repeated errors is smaller with than without "help".
4. This cannot be said for "help-help".
5. A mathematics student spends about 0.023 cpu hours per real hour, which means that, as far as the cpu is concerned, 30 students can simultaneously be connected without problems.
6. The number of errors made per hour seems to be constant: 15.
7. The number of errors made per command seems to be constant: 0.3.

Comparing the humanities students with the biology and mathematics students is not fair for the simple reason that the contents, the exercises and the size of the humanities course differed considerably from the biology and mathematics course. One can get an idea of the amount of work done in the workshop by considering that the biology students had to do 3 a-, 4 b- and 2 c- exercises while the mathematics students did 4 b- and 4 to 5 c-exercises. An a-exercise is very simple such as a procedure which prints the truth table of the operators "and" and "or". A b-exercise is more difficult, for example, a procedure computing a frequency table of letters occurring on an input file. A c-exercise is rather complicated as e.g. tic-tac-toe, simulation of Conway's game of life. The exercises for the humanities students were specially chosen from linguistics, such as text manipulation, e.g., counting letters, justifying text, coding and decoding text and generating text by means of syntactic rules.

### 5. ANALYSIS OF THE ERRORS

For each error, the attributes, as listed in table 2a, were measured for the three courses mentioned:

- n: frequency;
- h: the number of times "help" was called;
- w: the total time elapsed after the error occurred and until a "help" was called; it is called "1- help" wait time;
- hh: the number of times "help" was called twice in succession;
- ww: the total time elapsed after the error occurred and until the second "help" was called; it is called "2- help" wait time;
- e: the number of times the same error was immediately made again after one "help" was called;
- ee: the number of times the same error was immediately made again after two "help"s were called;
- r: the number of times the same error was immediately made again while neither "help" nor "help help" were called;
- t: the total time elapsed after the error occurred and until any reaction other than "help" or "list" was typed in; this time is called the think time.

table 2a. The total attributes.

From the above attributes concerning total/absolute quantities, we derived the following attributes describing relative quantities:

- mt: mean think time =  $t/n$ ;
- mh: mean number of "help"s =  $h/n$ ;
- mhh: mean number of "help - help" per one "help" =  $hh/h$ ;
- me: mean number of repeated errors after "help" =  $e/h$ ;
- mee: mean number of repeated errors after "help - help" =  $ee/hh$ ;
- mr: mean number of repeated errors after neither "help" nor "help - help";  $mr = r/(n-h-hh)$ ;
- mw: mean "1-help" wait time =  $w/h$ ;
- mww: mean "2-help" wait time =  $ww/hh$ ;

table 2b. The relative attributes.

The errors were sorted according to the above attributes, so that we obtained 17 lists of error numbers. It takes too much space to reproduce these lists here since each list contains 170 error numbers. We first give a few interesting examples and then we will describe the results after comparing these lists.

### 5.1. Some examples

In order of frequency,  $n$ , the 10 most frequently occurring errors were almost the same for the three courses: biology, mathematics and humanities; we therefore list the results of the biology course:

```

identifier not declared      (typing error)
syntax error
erroneous symbol            (typing error)
command expected            (typing error)
error in editor with text replacement
existing identifier expected (typing error)
    (as e.g. in list or edit)
variable did not get a value
editor reaches end of text too early
after editcommand no new line
error in string matching in editor

```

These 10 errors were responsible for 56% of all errors; four of them probably are typing errors, four are errors in the editor (which means that working with the editor is still not simple although we changed from an own-invented (clumsy) editor to the very handy UNIX editor) and two are more fundamental errors: one concerning syntax and one concerning semantics.

Errors which need the most time to think before a repairing action are the most important ones. They need considerable attention from the person who gives the lectures as well as from the makers of the system.

In the ordering of total think time,  $t$ , the first 10 errors for the biology students were:

```

variable did not get a value

```

identifier not declared  
 syntax error  
 command expected  
 identifier not declared  
 ", " or ")" expected after expression  
 too much cpu-time used  
 existing identifier expected  
 erroneous symbol  
 insufficient room left for array declaration

The mathematics students showed a somewhat similar behaviour; only three of the errors differed. The humanities students showed a more different behaviour, with four errors differing.

The reason that "identifier not declared" is showing up in the above lists is that it occurs so frequently, but it is, of course, a very common error. The most interesting errors are those which need the most mean think time, mt. The attribute mean think time is of interest as it says something about "difficultness".

The ten most "difficult" errors, then, for the biologists were:

insufficient room for array declaration  
 too much cpu-time used  
 division by zero  
 array identifier expected  
 "(" or identifier expected (in a command of the form "a:=  
 ...", where a is an array variable and "..." can be some-  
 thing like "(1,2,3)" or "b")  
 type identifier expected  
 (in "var a: ...")  
 index of one-dimensional array too small  
 erroneous use of standard procedure identifier  
 (as in "procedure sin(x,y:real)")  
 overflow of real capacity  
 first index of two-dimensional array too small

The frequencies of these errors range from 3 to 57, which on a total of about 10000 errors is of course very small. The think times range from 6 to 2.6 minutes.

According to the mathematics students the following ten errors were most "difficult":

too much cpu-time used  
 index of one-dimensional array too small  
 insufficient room for local array declaration  
 too much nested procedure calls (max = 50)  
 (problems with recursive procedures)  
 array identifier of same type expected  
 array-bound must be an integer  
 ("type ar = array [1..n] of real" is not allowed)  
 second index of array is too large  
 array index must be an integer expression  
 type of function must be standard  
 not implemented (probably an error of the kind: "a[1]:= b[1]"  
 occurring in a procedure where a and b are two-dimensional  
 array variables. This construction is allowed as a command,  
 but as it is non-PASCAL, we have forbidden it in a

procedure)

It is remarkable that 7 of these errors concern arrays. They were not made often; their frequencies range from 134 to 9 (which is not much compared with the total number of about 15000 errors) and their mean think times range from 10 to 2.3 minutes. With respect to the humanities students we remark that their ten most "difficult" errors differed completely from the above two lists, which is not surprising.

5.2. Some correlations

It is tempting to compare all the sorted lists of errors; the question is, however, how can they be compared? Heuristically, the most direct way is the way we suggested in the preceding section. Compare the first N items of two sorted lists. If the intersection consists of d elements, then d/N is a measure for correspondence. This number is called the correspondence number c(N) and is obviously dependent on N. If N is chosen to be 1, then c(N) is either 0 or 0.5; if N is chosen equal to the number of different errors (170 in our case), then c(N) = 1. We have computed c(N) for N = 10 and N = 20.

For the mathematics students we found the following correspondence numbers in the form of pairs c(10), c(20): (for obvious reasons we donot compare total attributes and relative attributes with each other).

	n													
n	1, 1	h												
h	.3, .4	1, 1	w											
w	.5, .6	.6, .7	1, 1	hh										
hh	.3, .5	.7, .6	.5, .7	1, 1	ww									
ww	.4, .6	.5, .5	.6, .7	.4, .7	1, 1	e								
e	.3, .3	.8, .7	.6, .5	.7, .5	.5, .5	1, 1	ee							
ee	.4, .3	.6, .6	.4, .4	.5, .6	.5, .5	.5, .6	1, 1	r						
r	.8, .9	.3, .4	.6, .6	.3, .5	.4, .6	.3, .4	.4, .4	1, 1	t					
t	.6, .7	.7, .7	.7, .8	.6, .6	.5, .7	.7, .5	.5, .5	.5, .7						

table 3. Correspondence numbers for total attributes.

The conclusion from table 3 might be that there is in general a rather high correspondence between all the total attributes, but in particular between n and r, w and h, hh and h, e and h, t and w. This conclusion is reinforced by a computation of Pearson's correlation coefficients. (not reproduced here).

The correspondence numbers for the relative attributes are given in table 4.

mt	mt		mh												
	1, 1														
mh	0, .2	1, 1	mw												
mw	.2, .2	0, 0	1, 1	mww											
mww	.3, .2	0, 0	.5, .6	1, 1	mhh										
mhh	.2, .4	.2, .5	0, .1	0, .1	1, 1	me									
me	.3, .3	.1, .1	.1, .1	.1, .1	0, .2	1, 1	mee								
mee	0, .1	0, .1	.1, .2	.1, .1	0, .2	.1, .1	1, 1	mr							
mr	0, 0	0, 0	.2, .2	.1, .2	.1, .1	.1, .1	.1, .1	1, 1							

table 4. Correspondence numbers for relative attributes.

The conclusion from this table is that not so much can be said: there seems to be a low correspondence between all of the relative attributes. The high correspondence among the total attributes is probably due to the fact that frequencies of the first errors of these lists are very high, whereas the frequencies of the first errors of the relative attribute lists are an order of magnitude smaller, so that stochastic effects on the order is much stronger. The only correspondences which can be noted are between mww and mw, which is not very surprising, and between mhh and mh.

The faint correspondence between mt on the one hand side and mww, mhh and me is notified.

Comparing two lists of 170 errors by counting how many errors appear to be common to the first N (N = 10 and N = 20 above) is of course very arbitrary. We could have taken the last N errors of the lists equally well.

Therefore, we also have worked out the following experiment. Compare the two lists by looking whether the i-th element of the first list occurs on one of the places i-d, i-d+1, ..., i+d-1, i+d in the second list. Such an element is called an OK element. A correspondence number can then be defined as the quotient of the number of OK elements over the total number of elements. For d = 10 we give here a list of 10 pairs of attributes which have the highest correspondence number:

mee-ee (70%)	tt-h (51%)
hh-ww (58%)	tt-w (51%)
tt-n (57%)	me-e (51%)
h-w (56%)	r-e (50%)
mww-ww (56%)	ee-e (49%)

The lowest correspondence numbers are between mee and mt (11%) and between mee and mh (13%). The highest correspondence numbers between relative attributes are for mhh and mww (40%) and mee and me (41%).

It is noteworthy that mt as attribute does not correspond well with any other relative attribute except maybe with mh (30%) and mw (23%). By means of Pearson's correlation coefficients, another correspondence between the relative attributes has been computed and the results are shown in table 5.

mt									
mt	1								
mh	.3	1							
mw	.4	0	1						
mww	.1	-0.2	.3	1					
mhh	-0.1	0	.1	0	1				
me	0	.1	-0.1	-0.1	-0.1	1			
mee	-0.1	.1	-0.2	-0.2	-0.3	.2	1		
mr	-0.1	.1	0	-0.1	0.1	0	0.3	1	

table 5. Pearson's correlation coefficients for relative attributes.

The positive correspondence between the mean think time (mt) on the one hand and mean number of "help" calls (mh) and mean "1-help" wait time (mw) on the other hand is interesting, although it is not evident that these attributes are really correlated. The observation that mee and mhh seem to be negatively correlated leads to the interesting conclusion that the more one does "help-help", the less one makes the same error again. That this conclusion, which certainly is a conclusion with which we would be very happy, should not be drawn too hastily, follows from the fact that a similar conclusion can not be drawn with respect to "help" (the correlation between me and mh even seems to be positive). The same correlation coefficient for the biology course, which was held earlier than the mathematics course, had the value +0.2. After this course was held the error messages for "help-help" were changed, this can be the reason of the negative correlation coefficient, discussed here. The positive correlation coefficient between mr (mean number of repeated errors without "help") and mee (mean number of repeated errors after "help-help") is notified.

6. THE DISTRIBUTION OF THE ERRORS IN TIME

From Kersten's [3] analysis we give a small account of the analysis of how the errors were distributed in time. Table 6 shows the percentages for six very frequently occurring errors during the three weeks of the course.

	week 1	week 2	week 3
identifier not declared	15.5	11.5	10.2
syntax error	7.6	9.2	7.9
command expected	4.7	7.5	4.2
existing identifier expected	4.4	3.8	5.2
variable did not get a value	1.7	4.3	7.6
error in editor	2.9	3.6	5.3

table 6. Time distribution of most frequent errors.

One can see that simple errors as "identifier not declared" are made less and complicated errors as "variable did not get a value" are made more as the course is going on.

## 7. ANALYSIS OF EACH INDIVIDUAL ERROR

With the error frequencies of the biology course we have analyzed each error individually with respect to: adequacy of error message and clarity of the location when the error occurred, and this with the relative importance of the error in mind. For about 20 errors this resulted in a better message and for three errors this resulted in reprogramming the pertinent piece of the system. Noteworthy is the error: "type of operands unequal" which occurs for example in "if i<10 and l<i", where "10 and l" is wrongly treated as a term; the reason for this was that we had a large table for all the operators and a three-dimensional "jump" such that knowing the types of the operands and the operator immediately led to the table entry where the definition of the operation was defined. We now have dropped this very clever scheme in favour of working out all cases with if-then-else and case-statements with the effect that errors can be better localized; as a side-effect it turned out that the new scheme was faster and needed fewer bytes.

It has been investigated also why about 30 errors did not show up in the statistics, although the system could produce them (by the way, there were 14 errors which occurred just once of the total number of 14800). The conclusion was that in principle they all can (and hence will) occur some time. Some constructs which are possible in BASIS are used very seldom, such as giving through procedure identifiers or call-by-reference parameters to other procedures. There is also a double syntax check: one is rather crude and concerns bracket structure, the other is precise. For example, the first check does not signal an error in "if 0<x<1 then ...", it sees that "then" has a preceding "if". The second check reads "0<x" as an expression and the next symbol to be treated is "<" so that an error is signalled saying that "then" is expected.

## 8. FREQUENCIES OF KEYWORDS

In order to observe what the students are doing and which constructs they are using we have counted their use of keywords as "begin", "if", etc., using the stat file. This has been performed, for several exercises, with the hope that per exercise the behaviour of the students is a little bit uniform. for the whole course it is very difficult to draw any conclusion about



the usage of keywords. Martin Kersten has performed such an analysis and could only draw conclusions like:

- "and" and "or" are used much more frequently than that boolean variables are being declared;
- "then" is used about twice as often as "else";
- the use of "for", "repeat" and "while" is proportional to 9:1:1.

This topic will be dealt with more extensively in a future paper.

## 9. THE INTERNAL FUNCTIONING OF THE SYSTEM

### 9.1. The internal structure and representation

In order to get an idea of the functioning of the system and thereby on the kind of measurements which have been carried out, we give the following global picture of the system and the internal representation.

The system is completely written in PASCAL.

The text of each BASIS procedure (i.e. typed in by the student) is kept in memory as a linear list of text cells. Each text cell contains a syntactic unit, such as an identifier or a keyword in which case a pointer in the text cell points to the character string constituting the identifier or keyword in the symbol table, or a number, in which case a pointer points to a record in the number table, containing the character string (for editorial reasons "123.456" is different from "1.23456e2") and the value, or the text cell contains a character like ";" or "=" or an end-of-line symbol, in which case backward and forward pointers simplify the process of stepping line-by-line through a text.

There is also an information store in which the information (type and for variables: value and for procedures: text pointer) about all identifiers is stored. There are pointers from the symbol table to this information store in order to find the appropriate information corresponding to an identifier occurring in the text. By a careful analysis of the frequencies of the records we were able to redesign the internal representation in such a way that the amount of memory needed for a certain program was diminished by a factor of about 2.5. Which means that a program can now have a size of some eight pages in the available space, which is more than enough for a basic course. For example, a text cell consisted in the previous system of 10 to 12 bytes, in the current system it consists of 4 (normal) to 10 (only for end-of-line cells) bytes. In the previous system we encountered the problem that the PASCAL system maintains 6 free lists of records which may be reused. The new system is provided with one record type: cell with 35 (nested) variants and the BASIS system itself maintains lists of free cells structured on size.

Another redesign concerned the symbol table. In the previous system it was organized as a binary tree upon initialization pre-filled with the keywords. It turned out that on the average 6 comparisons were necessary to insert or look after an identifier or key word. The storage structure of the current system is that of a hash table of moderate size of 248 bytes. The average number of comparisons now is 1.1.

The system code amounts to 155 PASCAL procedures totaling 3300 lines, having been thoroughly analyzed and judiciously rewritten

based on an analysis using counters and timing statistics of Martin Kersten [3]. The result was that the current system is about a factor 1.5 faster than the old one. We plan to describe the internal functioning of the system and the changes we made in more detail in another paper.

### 9.2. The hard- and software configuration

The hardware configuration consists of:

- PDP 11/45 with 124 K words
- cache memory (speed improvement about 40%)
- floating-point processor
- 2 RK05 disks (2\*2.5 M bytes)
- 1 fixed head disk (0.5 M bytes)
- 2 Ampex disks (2\*32 M bytes)
- Lineprinter + paper tape reader/punch
- 2 DEC Tape Units
- 30 Terminals
- 1 fast multiplexor for connection to a remote Cyber 73.

As software, the UNIX operating system [10] is in use; this is a time-sharing system allowing processes to share common files. The PASCAL compiler used for the BASIS system is home-made. It produces code in a very compact form, which is executed by interpretation losing about a factor eight in speed as compared to assembly code. The code for the BASIS system takes 26 K bytes and there are 18.5 K bytes necessary for library, tables and buffers. The system is itself also interpreting, so executing a procedure in BASIS is a time-consuming process. For example, an empty for statement of 100 repetitions takes 0.17 cpu sec and a while-statement 1.8 cpu sec. From the measurements in the next section we will see that executing procedures is done in only 9% of the total time, so the slowness is certainly not prohibitive. It is rather awkward, however, to see a student waiting for a long time while the system is executing his procedure, in particular at the end of the course. Therefore, a new system is under development, which combines the existing UNIX editor and PASCAL compiler to realize a more rapid BASIS system.

With regard to the PASCAL system we remark that a new system is almost ready in which the compiler itself takes 14 K bytes only. This system can be used to produce optimized short code or optimized assembly code. The ratios for speed and storage space of short code versus assembly code are roughly the same: four. The BASIS system compiled to assembly-code needs about 50 K bytes and runs two to three times faster than the current system. This BASIS system will be used during the next course so that actual measurements can detect improvements.

### 9.3. The response time of the system

Using the stat files it is possible to measure some interesting response times. In a typical conversation there are certain times which are of interest. They are shown in fig. 1.

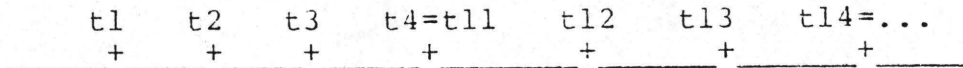


fig. 1 Events during a conversation.

The times are defined as follows:

- t1, t11: student starts typing a line of text;
- t2, t12: student sends line to the system;
- t3, t13: BASIS has seen the line and sends the line, provided with current real time and current cpu-time, to the stat file;
- t4=t11, t14: the computation as specified by the line is performed and the reaction of BASIS is sent to the terminal, whereupon the student can start thinking and/or typing a new line thereby repeating the cycle. If the system reports an error this is put, together with the current times, on the stat file.

Evidently, the time  $t4-t2$  is the response time which we want to measure as a function of the number of other simultaneous users of the system.

The following experiments have been carried out.

First, we asked  $n$  BASIS users to edit only, typing a line now and then. For  $n$  running from 0 to 18 this had no effect on the response time of the  $(n+1)$ -st user.

Second, we asked  $n$  BASIS users to use the system heavily by executing an infinite loop, again for  $n=0, \dots, 18$ . The effect on the response time of the  $(n+1)$ -st user was measured as follows. The  $(n+1)$ -st user was executing a procedure which used about 1 cpu second. The real time was measured this procedure needed to get ready. This was done by computing  $t13-t3$  and neglecting  $t12-t11$  (using the UNIX facility to type one line ahead). The results can reasonably accurately be described by the formula:

$$r = 1.7 (n+1),$$

where  $r$  is the ratio of real time and cpu time.

The ideal formula would have been  $r = n+1$ ; the factor 1.7 is caused by swapping overhead.

The response time for just typing in a new line could not be measured by means of the stat file, since in  $t3-t2$  an unregistered amount of time elapsed before BASIS turns attention to the user. This time,  $\delta$ , could be measured by simply using a stopwatch. It is given in the following formula:

$$\delta = 0.2 (n+1) \text{ sec.}$$

It is of course also interesting to see from the stat files how long a student really waited in the course for the execution of procedures. Therefore, we computed  $t4-t2$  from  $t4-t2 = (t13-t3) - (t12-t11)$  assuming that  $t13-t12=t2-t1$ .

The time  $t12-t11$ , i.e. think time plus type time, was estimated using cases where the procedure call lead to a syntax error.

The following conclusions could be drawn; they are given in table 8.

	week1	week2	week3	total
total real time waiting for computation (in min)	8	15	129	152
percentage of real time waiting for computation compared to total time connected (in %)	1	5	14	9
total cpu time for procedure computation (in min)	1.7	2.6	15	19.3
percentage of cpu time needed for procedure calls compared to total cpu time used (in %)	50	55	63	62
ratio of real time waiting for computation and the cpu time used for it	4.7	5.8	8.6	7.9

table 8. Response- and waiting times for procedure calls.

The conclusion from table 8 is that during the third week of the course on the average 2.25 students of the 15 simultaneous students were using the system heavily. This would result in a ratio of 3.8 for real time over cpu time a procedure call needs. The actual observed ratio was 8.9. An explanation for the difference is that firstly the students which are editing are using the system more heavily than in the experiment above. Secondly, while the course was going on an unknown number of other people were using UNIX. So the actual ratio of 8.9 is quite reasonable.

#### 10. FUTURE PLANS

As has been said already in section 9.2, a new BASIS system which is built around the existing editor and existing compiler is under development

Another plan is to direct the measurements on the individual behaviour of the student: is he making too much errors so that he needs help, is he using strange constructs. This is, however, quite complicated since it is then necessary to know the behaviour of a "normal" student. At the moment we only save his last ten errors in order to be able to give him a message when he makes the same error more than two times in a certain time period.

#### ACKNOWLEDGEMENT

The author is very grateful to Ruud Wiggers, for doing all the programming and the measurements and to his student Martin Kersten for numerous ideas and critical remarks. Furthermore, he thanks Anthony I. Wasserman, University of California, San Francisco, and temporarily guest of the vakgroep informatica, for numerous remarks concerning the text of this paper.

#### LITERATURE

- [1] K. Jensen, N. Wirth, PASCAL User Manual and Report, Lecture Notes in Computer Science, Springer, 1974.
- [2] John D. Gannon, Language Design to Enhance Program Reliability, Technical Report CSRG-47, January 1975, Computer Systems Research Group, University of Toronto. Also available in SIGPLAN Notices 10,6.
- [3] M. Kersten, An Evaluation of the BASIS System, Informatica Report IR-21, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1977.
- [4] Donald E. Knuth, An Empirical Study of FORTRAN Programs, Software - Practice & Experience, Vol 1, No. 2 (1977) pp 105-134.
- [5] R.P. van de Riet, Some Criteria for Elementary Programming Languages, in Computers in Education, O. Lecarme and R.W. Lewis (Eds), IFIP, North Holland Publishing Company 1975, pp 953-963.
- [6] R.P. van de Riet, BASIS - an Interactive System for the Introductory Course in Informatics, Information Processing 77, Proceedings of IFIP Congress 77, North Holland Publishing Company, 1977, pp 347-351.
- [7] R.P. van de Riet, R. Wiggers, The Implementation of BASIS, Report IR 13, Wiskundig Seminarium, Vrije Universiteit, Amsterdam, 1976.
- [8] M.E. Sime, T.R.G. Green, D.J. Guest, Psychological Evaluation of Two Conditional Constructs used in Computer Languages, Int. J. Man-Machine Studies (1973) 5, pp 105-113.
- [9] A.S. Tanenbaum, Implications of Structured Programming for Machine Architecture, Communications of the ACM, Vol 21, No. 3, (March 1978), pp 237-246.
- [10] K. Thompson, D.M. Ritchie, The UNIX Time-Sharing System, Communications of the ACM, Vol 17, No. 7, (July 1974), pp 365-375.



Walter Volpert/Reinhard Frommann

Berlin.

Software assessment from the viewpoint of the psychology of action



The research on "human factors of computing" should not be reduced to the mere problem of increasing efficiency. The individual and social effects of information technology are pointing to the question whether the essential aspects of human thinking and action are considered in man-computer-interaction.

If this has to be denied, the attention must be focused to the question how information technology affects human thinking and action, e.g. taking away its independence and creative power.

Some important aspects of this subject will be discussed.

Industrial psychology  
- nicht allein verstehen abschließen  
v. C-M-Interaktion





## PROGRAM DEVELOPMENT STUDIES BASED ON DIARIES

Peter Naur

Copenhagen University

doel: simulator in FORTRAN  
 Tabel of control  
 INTEL-

\* sein claims.  
 ondogmatic notes  
 \* notes sein program  
 ming

## Abstract

The notion of program diaries used as basis for studies of program development processes is introduced. As illustration a particular program development is discussed. The study yields results related to the use of formalization in problem analysis and to errors in programming. It is suggested that a form of problem analysis that requires every part of the program to be explicitly justified may lead to error-free program design.

1. Introduction

The purpose of the present notes is to discuss the use of diary notes for illuminating the problems of program development. By diary notes will be understood here notes describing problems considered and solved in the course of the development of programs, taken day by day as part of the development process itself. The aspect of such notes considered here is their recording of what actually goes on in the programmer's mind during programming, in the sense of what the programmer perceives to be the task to be done and the problems and their solution. Upon subsequent analysis, diary notes may contribute to an understanding of the problems of programming, and to developing effective programming techniques. For earlier experiments in this direction see Naur (1972) and Naur (1975).

The study of programming diaries is seen here as a supplement to other kinds of studies of the programming process, in particular group experiments in which the behaviour of several individuals facing the same, constructed programming task is studied (see e.g. Brooks, 1980). While such experiments are indispensable in ascertaining the general validity of relevant hypotheses, less structured, empirical work, such as studies of diaries, seems important in finding what hypotheses might be worth investigating for validity.

An essential feature of the diary approach, as understood here, is that the program development reported on in the diary should be a true performance presenting, in some of its aspects, new problems to the programmer. This requirement is imposed so as to exclude from consideration such descriptions of program developments that appear to depict real events, while in reality they are for the most part constructed so as to give credence to a given methodology. In the kind of diary considered here one would expect to see displayed both unfruitful attempts and unexpected successes.

It must be recognized that in the requirement that the development presents new problems to the programmer lies a deep problem of the approach, namely to identify that which can be admitted as new. In principle the diary should start with a complete enumeration of everything the programmer knows already and that might be relevant to the problem at hand, so as to make it quite clear to what extent the problems encountered in the development can be solved by means of techniques that are familiar to the programmer.

A complete enumeration is clearly practically impossible. In practice the background knowledge can only be stated to some limited extent, while usually much of it will have to be inferred indirectly from the description of the problems encountered and their solution.

In the present notes the approach is illustrated by only a single instance of a diary. This happened to be available for study, having been produced by the author for a different purpose, and thus describes a program development as it proceeded without regard to any subsequent analysis.

## 2. Programming of a well-known task

As illustration of the use of a diary in programming studies, this section will review some results obtained in a concrete, modest programming task. The outlines of the task are as follows:

1. Purpose of programming task: to develop and document a simulator, expressed in Fortran, for the microcomputer INTEL 8080. The ultimate purpose was to establish a model solution of the problem, for use in university teaching.

2. Background of programming task: in addition to the specifications of the computers and programming languages involved, of the format of the simulator input language, and of the essential requirements on the simulator output, it was required that the simulator should be designed around a table controlling the analysis and simulated execution of the microcomputer instruction words. The relevant programming techniques were very well known to the author from several earlier similar program development tasks.

3. Program development: as the most unusual feature of the development process, all substantial design considerations and decisions were recorded as the work progressed as a typed problem analysis report. In this manner the basis of each part of the program was established in writing before the actual programming was done. Within the total development three subphases can be distinguished, although they overlap to some extent: (1) design of instruction word analysis and central simulator actions; (2) design of control by table; (3) design of output format.

4. Program punching, testing, and correction: the interface to the computer executing the simulator was given as a conventional batch-oriented operating system, with a turn-around time of the order of 15 minutes, and with primary input from punched cards.

5. Design and verification of tests: a properly documented test of the simulator was a vital part of the task. For testing a series of 16 INTEL 8080 test programs was designed and their correct execution in the simulator verified. The development of the test programs and the testing and correction of errors of the simu-

lator itself were done hand in hand. In this manner each run with the computer would usually include many independent test executions, corresponding to the various test programs, thereby allowing a very productive utilization of the batch operation mode.

From this development the diary to be studied is formed as the collection comprising the problem analysis report, test notes, programs, and computer output, together with a record of the time spent on the project day by day. The magnitude of the total task is summarized in table 1.

Table 1. Magnitude of programming task

Work phase	Time spent		Lines produced		
	Calen- dar days	Work hours	Free text	Comments in pro- grams	Pro- gram
1 Design of program, writing of problem analysis and program text	24	38.5	666	154	679
2 Punching, proof reading	2	10.6	0	0	0
3 Writing of test notes and programs, doing test runs and correc- tions of errors	16	31.5 <i>justify tests!</i>	251	0	<u>369</u>
Total	42	80.4	917	154	1048

For the purposes of the following discussion the nature of errors found during the testing is particularly pertinent. For this reason the outline of the testing history is given in table 2, and an overview of the corrections made in table 3.

### 3. Program corrections

Table 3 gives an overview of all corrections to the simulator program and to the test programs made as a result of the program testing. In the following notes these corrections will be discussed, with special attention to the psychological issues that seem relevant to an understanding of them.

The most conspicuous overall feature of table 3 is the wide difference in the number of corrections in the various groups. What will be argued here is that these differences closely reflect the author's conscious judgement of the importance of each group, in the sense that the larger numbers of corrections are found in groups that the author knows full well are less vitally dependent on correctness. This rule is seen confirmed in several ways. First, the most numerous group of corrections is 3, errors of program text format, which is concerned purely with the appearance of the printed program text, and which in no way influences the operation or correctness of the program. Second, the second group according to size

Table 2. Testing history

Run 1, 1980 June 4, 11.07. The Fortran compiler referred to in the operating system was obsolete and produced large numbers of spurious error messages.

Run 2, 1980 June 4, 16.50. Diagnostics: from Fortran compiler: none; from loader: one undefined name (indfad, used for infad).

Run 3, 1980 June 4, 17.02. No diagnostics. Execution of one minimal test case, T20.

Run 4, 1980 June 4, 17.18. Execution of six minimal test cases, T21 - T26. All fail because of the same error, viz. destruction of argument in central input conversion function.

Run 5, 1980 June 9, 11.33. Execution of seven minimal and two productive test cases, T1-T2. Essentially correct execution. Errors in details of output format.

Run 6, 1980 June 9, 12.57. Execution of new test case, T3, leads to infinite loop, because of error in test data.

Run 7, 1980 June 10, 14.12. Execution of seven minimal and five productive test cases, T1 - T5. Everything worked correctly.

Run 8, 1980 June 11, 13.57. Execution of six new test cases, T6 - T11; no errors found.

Run 9, 1980 June 17, 15.36. Error in operating system control instruction.

Run 10, 1980 June 17, 16.55. Execution of five new test cases, T12 - T16. Four had trivial errors. No error in simulator found.

Run 11. 1980 June 19, 13.57. Correct execution of four corrected test cases, T12 - T15. No errors in simulator found.

Table 3. Corrections to simulator program and test programs

Group	Test run in which error was noted }	Number of corrections						
		2	5	6	9	10	11	Total
1 Error of simulator logic		3	1					4
2 Misspelt name		1						1
3 Error of program text format		14	1					15
4 Error of simulator output format		1	4					5
5 Error in test program			1	1	1	4	2	9

is 5, error in test program, which shows 9 corrections to the 369 lines of test program. This may be compared with the total number of corrections to the simulator program itself, groups 1, 2, and 4, with altogether 10 corrections to 679 program lines, of which the 5 corrections of group 4 are concerned only with the relatively inconsequential matters of output format. It seems quite clear that when working out the test programs the author's awareness that errors entering at this stage would have only slight consequences has led to a much increased error rate. Third, as already noted, of the total number of corrections to the logic of the simulator program, groups 1, 2, and 4, half come from the relatively unimportant group 4.

Considering now the core errors of the simulator program, corresponding to the 5 corrections of groups 1 and 2 of table 3, it is remarkable that none of them has been found later than test run 5. As seen from table 2 this means that the work on test programs T3 to T15 did not reveal any further errors in the simulator. This

observation becomes still more remarkable by a closer inspection of the details of the errors behind the five corrections. One was a spelling error, detected by the loader as an undefined name. Three errors were detected fortuitously by inspection of the simulator program. Only one error was found by its influence on the execution. Every one of these errors was a purely local slip of the pen or mind. In other words, every part of the simulator, from the overall plan, via the control table, to every detail of the instruction word analysis, execution and addressing, worked perfectly according to the design, without any modification or correction whatsoever.

In view of the general interest in program correctness and the attention given to it in works on programming methodology, this positive result of the present program development will be further looked into in the following section.

~~maintain~~  
~~intuitive~~  
informal?

#### 4. Problem analysis work mode

As background for the analysis of the problem analysis that led to a flawless program design, some knowledge of the underlying author attitude may be illuminating. This attitude has, as one important component, the view that since programming must finally rest on the programmer's direct, intuitive understanding, the criteria for the program produced being "right" must be based, ultimately, on the programmer's looking at what he or she has done and accepting it. From this view it follows that for helping the programmer in his or her task, what is important is anything that may make the programmer retain his or her alertness in the face of the mass of detail constituting the program, and that may make every one of these details relevant to his or her direct understanding. The ideal situation is one in which the programmer may take any part of the program, look at it, and decide that it is right or wrong. Where this ideal cannot be realized because the algorithmic means available in the programming language are too remote from the actions that the programmer directly sees to be required for solving the problem, the programmer will have to bridge the conceptual distance by means of a suitable intermediate platform of concepts. This, then, is the task to be done in the problem analysis: the programmer must consider each aspect of the problem in turn, and for each decide whether it can be realized directly by programming or whether a conceptual bridge is required, and in the latter case, he or she must build the bridge.

As the work mode for accomplishing this task the present program development has used a technique of problem analysis according to which every part of the program is developed through a conscious process, the steps of which are recorded, as far as possible, as a fully articulated written argumentation. In this process the development of each part of the solution is allowed to proceed along its own path, informal or formal, restricted only by the one basic guiding principle, that every part of the documentation must be made to appear such that the validity of the underlying argumentation is intuitively obvious. As shown in table 1, in the present case this manner of developing the program has led to the formulation of 666 lines of problem analysis text, which is roughly the same number of lines as in the program itself. While most of these lines are informal prose, several different kinds of formalized expression have been

Table 4. Formalizations in problem analysis

Kind of formalization	Number of lines
Simple list of target machine instructions	30
Structured list of target machine instructions	60
Target machine store map	13
Arithmetic formula (address calculation, etc.)	8
Target machine word position display	3
Algorithm fragment	4
Control word format description table	16
Case enumeration table	17
	<hr/>
Any formalization, total	151

employed, as summarized in table 4. What is most noteworthy of this table is the fact that so many different kinds of formalization have been found useful. Since clearly each such kind can only be used as a result of a deliberate consideration of the manner in which a particular part of the solution is most effectively described, the employment of these several kinds indicates that in the manner in which the problem analysis is carried out the choice of the most suitable mode of expression appears prominently.

The insistence that the problem analysis provide an intuitively obvious justification of each part of the program implies that the problem analysis effectively includes a proof of the validity and correctness of the program. In the present problem no explicit proof of any part has been given in the problem analysis, the solution having been proven correct by construction. This fact is a reflection of the nature of the problem. In solving other problems the same general approach might very well lead to a need for demonstrations that depend on such intermediate steps that are characteristic of proofs. Such demonstrations can be accommodated within the present frame of work without any difficulty.

In addition to the arguments for the correctness of the solution, the problem analysis must include arguments that justify the choice of each part of the solution. These arguments may be expressed in any appropriate manner, and in particular may include discussions of effectiveness and of alternative solutions.

More generally, the mode of work employed in the present program development may be said to conform closely to that employed in normal technical activity concerned with systematic development and construction.

## 5. Results of the diary analysis

The diary analysis given as illustration above suggests, as one result, that when the programmer is fully aware of the implications of errors in various parts of a project, his or her error rate will be influenced strongly by the severity of the consequences of errors in each part, in the sense that more errors will be made where they matter less. This result accords well with common sense and with experience reported earlier (Smith 1967).

As another result, a problem analysis aiming at a full justification of every program part in the most effective manner has resulted in an analysis report written mostly in prose, but making use of eight different kinds for formalization. This result may be viewed as a sceptical comment on some recent work aiming at establishing a single formalized notation for program specification (see, e.g., Liskov and Zilles, 1977).

The most striking result of the present study is the indication of a high level of program design correctness obtained by a problem analysis requiring a written, articulated justification of every part of the program. This result, if generally valid, would be highly important to practical development. It therefore suggests further studies aiming at clarifying whether a problem analysis of the kind considered is a feasible approach to solving at least some of the problems of error-free program construction. Such studies should aim at finding out to what extent errors in programming are related to the argumentation used when writing the program. As another aspect, the further studies might attempt to yield observations related to other persons and problems. In either case the studies might very well make use of the diary approach, and indeed, it might be difficult to pursue them in any other way.

In conclusion it appears that several important aspects of programming can be conveniently and effectively studied by means of diaries, in the sense described above.

#### References

- Brooks, R.E. 1980. Studying programmer behaviour experimentally: the problems of proper methodology. *Comm. ACM* 23 (4): 207-213. \*
- Liskov, B. and Zilles, S. 1977. An introduction to formal specifications of data abstractions. In *Current Trends in Programming Methodology*, Vol. 1, ed. R. T. Yeh, pp. 1 - 32. Englewood Cliffs, New Jersey: Prentice-Hall. |
- Naur, P. 1972. An experiment on program construction. *BIT* 12 (3): 347-365.
- Naur, P. 1975. What happens during program development - an experiment. In *Systemeering 75*, ed. M. Lundeborg and J. Bubenko, pp. 269-289. Lund, Sweden: Studentlitteratur.
- Smith, W.A. 1967. Nature and detection of errors in production data collection. *Proc. AFIPS 1967 Spring Joint Computer Conf.:* 425-428.

Peter Naur  
Begoniavej 20  
DK - 2820 Gentofte

Gause  
Formalisation Program  
Designers correctness  
~~File~~

Ulrich:  
conviviality

(also in comp)

Xerox Star system

knowledge Base

- communication
- problem domain
- partner



verbreden

- main systemen ↔ <sup>met</sup> connect-iv
- help systemen: meer alle informatie
- visualiteit
- manipuleren concrete object

Byte-Map ← Smalltalk

objectalk ←

• ~~screen as BISH~~ (Franz Lisp)

↑  
Berkeley



INSTITUT FÜR INFORMATIK  
H.-D.Böcker/G.Fischer/R.Gunzenhäuser

UNIVERSITÄT  
STUTT GART

Institut für Informatik, Azenbergstr. 12, 7000 Stuttgart 1

Azenbergstr. 12  
Herdweg 51  
7000 Stuttgart 1  
Telefon (0711) 2078-  
Telex TX 07-21703

March 16, 1982

A b s t r a c t  
=====

H.-D.Böcker / G.Fischer / R.Gunzenhäuser

Project INFORM: The function of integrated information manipulation systems (IMS) to support man-machine-communication

Goals of the project:

- Critical evaluation of current efforts to build IMSs
- Development of a requirement analysis for an IMS
- Design and implementation of a prototype for an integrated knowledge-based IMS
- Empirical investigations of our prototypical IMS with respect to user interface, user behavior, partitioning of cognitive tasks between humans and machine, and use of an IMS as a learning and working environment.

The application areas for an IMS considered within INFORM are

- software engineering/software development systems
- office automation systems.

Theoretical base:

Our project is based in Artificial Intelligence and Cognitive Science Research. Representation of knowledge, understanding the cognitive capabilities of the user, design support systems, question answering systems, support for exploratory programming, uniformity of system behavior across many domains are some research fields playing an important role in our work.

Abstract

Software:

The software to be implemented within the project consists of basic tools (e.g. window systems, knowledge representation mechanisms) as well as application packages. The latter are of two kinds:

- 1) A planning system supporting students planning their graduate studies. Through the use of domain specific knowledge the system helps the student by pointing out consequences of decisions, and allowing the exploitation of alternative designs;
- 2) A system providing assistance in setting up a finance plan for a research project; this subtask concentrates on the propagation of constraints and how to make them visible to the user.

Our intention (in contrast to some work in AI) is not to build fully automatic systems, but symbiotic systems between man and machine.

II. FACILITATING HUMAN-COMPUTER INTERACTION

A. Tools and aids



COMPUTER AIDED DECISION MAKING WITH GRAPHICAL DISPLAY OF  
INFORMATION

Bernard SENACH

Institut National de Recherche d'Informatique et d'Automatique  
Domaine de Voluceau - B.P. 105 - 78153 Le Chesnay - France

ABSTRACT

These studies have been conducted in a railway transportation system and are centered on problem structuration activity. Skilled operators do not apprehend the problems to be solved with all the accurate information: they simplify it and some variables are not processed. This cognitive difficulty is stated in terms of **problem space reduction** and related to the displayed information in order to define the general objective of a computer aided decision making system.

1 - Introduction

The main problem with man machine communication is the **compatibility of two representations**: the machine representation as a product of the designer's choices and the operator's representation as a result of processing of the information displayed.

According to that perspective, the aim of ergonomics is to design beforehand the machine in such a way that it is fitted with the subject's cognitive representation (Bisseret and al., 1979).

The most frequent way to proceed is to design the machine from this cognitive study (see for example Falzon, 1982).

Another way of looking at it is to find out if the information displayed can induce an inaccurate cognitive representation.

As we will see, this situation can be rephrased in terms of **problem space reduction** in order to define the general objective of decision making systems.

2 - The man machine system

The studies presented here have been conducted on the PARIS' subway. Our research concerns the design of a new control room, including the computer aided regulation system.

The most important task to be performed is to solve regulation problems, with the aim of reducing the incidents' repercussions: the operators have to maintain passengers transportation despite traffic interruptions.

### 3 - The man machine analysis

The man machine analysis was completed in three steps.

#### 3-1. Wholistic approach

A first step was a wholistic approach including usual job analysis' tools and techniques: observations, interviews, critical incidents analysis (Senach, Janet, 1979; Senach, 1980a).

#### 3-2. Formal definition of the problems structure

The second step was centered on a critical issue for a computer aide system: the problem solving activity. The objective was the identification of formal features that would distinguish problem classes and types of incidents, so that:

- all incidents showing the same pattern features belong to the same problem class and define a single problem for the operators, whatever are the other differences in their surface description.
- two incidents different according to a formal feature belong to two different problem classes, and consequently the operator cannot use the same procedure to solve these two incidents.

The difficulty in identifying these formal features concerns the system complexity: two incidents can be described by very different patterns (not the same variables, and if the same, different values).

An experimental problem solving simulation was run, using a withheld information technique.

The subjects had to solve incidents by asking the information to the experimenter, and each question was recorded. Eight skilled operators solved six problems (Senach, 1980b).

### Results

The controlled objects are bi-dimensional:

- one dimension refers to the planning of the train movement
- the other dimension refers to the drivers assignment.

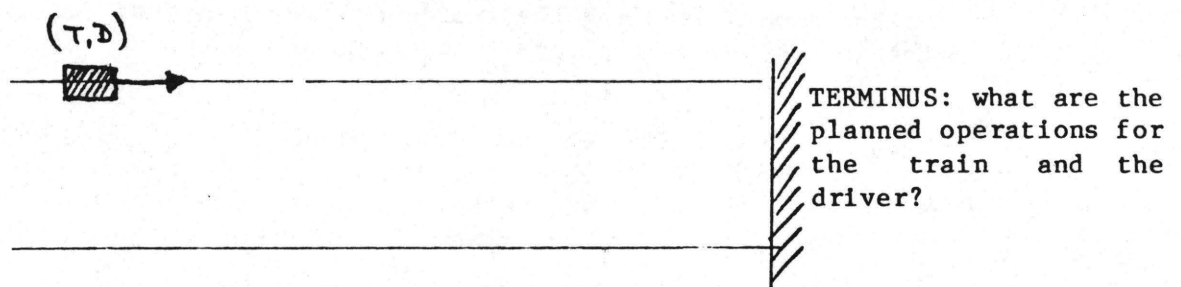
These two dimensions are independent: a driver does not always stay with the same train all day long.

Therefore, the problem classes can be defined according to the structure of the train-driver pair.

As a fonction of what has been planned for the driver and his train the repercussion, and thus the problems complexity, may be very different.

Here are two examples of possible future states and of related problem:

A train (T) and a driver (D) have an incident:



Case 1:

The train had to go on a siding.  
The driver had to be out of duty.

There is no repercussion on the other line.

Case 2:

The train had to go on a siding.  
The driver had to take a new train.

The new train will be late, and the operator in the control room has to find a new driver for the new train (5 different elementary cases may be distinguished).

3-3. Problem space reduction

The **third** step further investigated a particular point: the previous experimental data showed that the skilled operators do not apprehend all the relevant information necessary for the problem solving.

This lead us to the hypothesis of a cognitive difficulty in problem solving that could be expressed in terms of **problem space reduction**.

This notion has already been used in problem solving litterature (Newell and Simon, 1972) but generally in another acceptation.

For instance, Simon and Reed (1976) stress that when the problem constraints are well defined the problem solver has to evaluate few alternatives and the task environment tends then to reduce the space in which the search has to take place.

The problem solver may even more reduce the space using a strategy: Elstein and al. (1978), pointed out that in complex systems - medical diagnosis- when the potential size of the problem space is important, the operators have to reduce it: early hypothesis are generated in the very first minutes of the meeting with the patient.

The meaning of the reduction is here quite different.

The general idea is that, even though skilled, a subject does not identify and use the relations contained in a problem structure which are necessary in order to solve the problem.

In other words, the operator filters the displayed data and then may, in a complex system, process a problem simpler than (or different from) the real one.

Some clarification about the reduction mechanism has been suggested by Richard (1972): during problem analysis, if a subject cannot refer to any schemata guiding the identification of critical features he cannot use planning; as soon as an operational representation is reached (1), analysis activity stops and execution starts.

This description concerns what happens when the problem is really new for the subject, but can be extended to experts by referring to the operators' difficulties.

In the existing regulation system at the beginning of an incident the operators have to make a diagnosis and evaluate the repercussions, i.e. they have to identify which operations were planned for both the train and the driver and what are the consequences if these operations are not fulfilled.

The origin of the problem lays in short term memory limitations: it is well known that it is very difficult to make inferences on two dimensional objects.

The difficulty is increased because of the displayed information. One of the two dimensions is more salient than the other:

- train positions are displayed in real time on a control panel and the most important document is a graphical representation of the process showing all the theoretic states and trains position.
- data about drivers are only supplied on an alphanumeric list providing for each driver all his assignments.

So the data are not organised as paired structures. Some of the information is gathered by the operators: they write it down, but it is not always reliable.

### Experiment

An experiment intending to test the reduction hypothesis was carried out (Senach, 1982). More precisely we tried to show that the information about the drivers is not systematically processed by the operators. A problem solving simulation was used again.

Four incidents structurally different according to our formal features had to be solved under two experimental conditions:

1. Subjects solved the four problem using the existing tools.  
**then**
2. The same subjects solved the same four problem with a new information display.

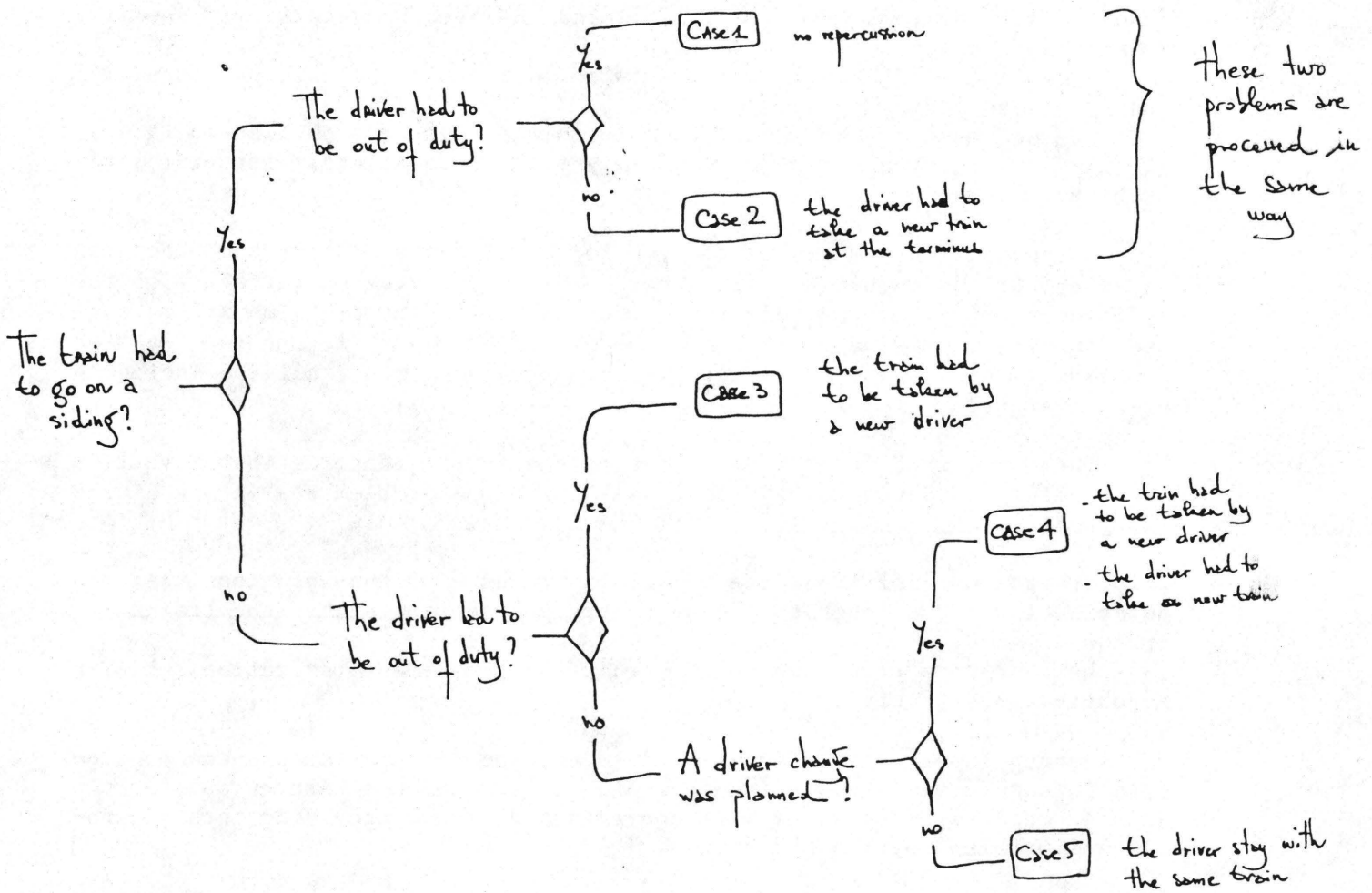
Five skilled operators took part to the experiment.

**The new information display.** Its structure was quite the same as the existing tools structure. The only difference concerns some critical properties of the problem not apparent on the usual display, that were pointed out (variables relations, order constraints between the trains...).



Results

The following figure shows that some of the variables are not processed:



The very interesting result is that in case 2 none of the solution at the first problem presentation can make up for the delay of the new train.

There is a diagnosis error. In terms of information processing theory, the problem space reduction can be described as a decision tree that would have fewer branches than the theoretical one: some of these branches are not used and might even not be known, and some given branches may not be completely processed.

In other words, if the diagnosis is defined as the result of a categorization process, the reduction means that the whole set of possible diagnosis has not been built by the subjects or that the categories are defined in such a way that two different problem can be processed in the same way by an operator.

#### 4 - Conclusion

1. A first outcome in formulating these problem is to point out that "usual" job analysis may have some limits. Skilled operators are generally treated as experts.

Ergonomics then rely on the knowledge of their operations, procedures and/or strategies.

But we have shown here that in complex systems, skilled operators, having several years of experience may have not completely structured the problems.

2. A second outcome is that, even for a skilled operator, each incident may appear as a new one. In other words, the degree of certainty of the efficiency of a given procedure is low: two different problems are processed in the same way, and the procedure is not so efficient for the both problems. This results in that, in the operators' mind, all the incidents are different.

3. The computer aided system relies on the formal features that have been identified: up to now we have only dealt with the problem analysis activity (diagnosis).

In **problem solving**, the general structure of the solutions is a **substitution**: the operators have to try and replace either the trains or the drivers.

The operators have then to construct the possible substitution set with a combinatorial activity.

The new information display used in the second experiment seems to provide in some cases a good diagnosis aid, but it was not kept as an effective tool because there was a more convenient display, providing both diagnosis and problem solving aiding.

This last display can be described as follows:

- According to which problem class belong an incident, the information needed to solve it is not the same: what are the available replacement trains and/or drivers?
- It is possible to precisely define the meaning of "available" from the analysis of experts decision making. The analysis of the choice criteria between several possible substitution elements results in a solution hierarchy.

Having defined basic tool structure, it is the necessary to clarify the automation level. Several options are possible: from the simple information display (data), to their combination (solutions), and up to an expert system.

REFERENCES

- Bisseret, A., Michard, A., Boutin, P. (1979)  
Eléments introductifs à l'ergonomie des systèmes Hommes-Machines.  
Informatique et Sciences Humaines, n° 44.
- Elstein, A.S.; Shulman, L.S.; Sprafka, S.A. (1978)  
Medical Problem Solving, An Analysis of clinical reasoning. Harvard  
University Press. Cambridge, Massachusetts.
- Falzon, P. (1982)  
Displays structures: compatibility with the operators' mental repre-  
sentation and reasoning process. Proceedings of the 2nd European  
Annual Manual, June 1982, Bonn.
- Newell, A., Simon, H. (1972)  
Human Problem Solving. Englewood Cliffs, Prentice-Hall.
- Richard, J.F. (1982)  
Le traitement humain de l'information: Sa contribution aux sciences  
cognitives. Communication au Colloque de l'Association pour la  
Recherche Cognitive, Février 1982, Pont à Mousson.
- Senach, B.; Janet, E. (1979)  
Propositions pour l'aménagement du Poste de Commandes Centralisées  
de la ligne de Sceaux. INRIA RER 7910 R01
- Senach, B. (1980a)  
Analyse du travail de contrôle d'un réseau ferré: Recherche des ina-  
daptations du système Homme-Machine. INRIA RER 8005 R02
- Senach, B. (1980b)  
Analyse du travail de régulation d'un réseau ferré: Résolution d'in-  
cidents d'exploitation. INRIA RER 8012 R05.
- Senach, B. (1982)  
Aide à la résolution de problème par présentation graphique des  
informations. INRIA Mars 1982, n° 13.
- Simon, H.A., Reed, S.K. (1976)  
Modeling strategy shifts in a problem solving task. Cognitive Psy-  
chology, 8, 86-97.



# The Case for Control Independence in Dialogue-oriented Software

-- Draft version --

by

Sture Hägglund

Software Systems Research Center  
Linköping University and Institute of Technology,  
S-581 83 Linköping,  
Sweden

**ABSTRACT:** Designing the human-computer interface in interactive systems is a task considerably different from designing algorithmic programs for computation. Thus we should have language constructs and employ software architectures, which recognize these differences when implementing dialogue-oriented software. This paper discusses the concept of control independence, as a basis for realizing software where dialogue control is completely separated from internal computations. In this way modelling of multi-style dialogues can be supported. Some experiences with tool systems, which implement this approach are reviewed.

- prompt  
- menu  
- screen  
- command forms  
  • par + keywords  
  • par • position  
user-selected!  
Acceptable messages  
- simple compound  
\* context dependent interpretation  
• possible transitions between context

description der claus für inter-  
prozedur wird.  
login

a ttr  
buteo

## 1. INTRODUCTION.

Computing devices were originally thought of as data processing machines, where input is fed into the system and computed output delivered as a result. Characteristic for this view is that the computer system implements a mechanism for transforming input to output according to some algorithm. Parameters for the processing are elements of data, the management of which is considered secondary to the computations.

However, as data processing applications matured, the focus of interest were gradually shifted in favour of the data management aspects. Computers became more and more thought of as instrumental for handling repositories of data. Under this view, processing of the data is but one function available when storing, organizing and retrieving information. Database systems are becoming essential as parts of the software environments provided for implementation of various computerized services. The same evolution is reflected in the development of higher-level programming languages with the introduction of support for data abstractions, object-oriented computations and data-driven programs [LIS77, ING78].

-156-

The current development seems to emphasize more and more the central role of *communication* in connection with computer utilization. This fact is manifested by the increasing importance attributed to internal cooperation between different software systems or physical communication in distributed computer networks, as well as the need to understand and implement efficient human-machine communication using the computer as a responsive tool for human problem solving.

This discussion brings us to the central topic of this paper. It appears that there are many cases, when a software system should be viewed as a realization of a communication system for an information repository where different kinds of data processing tasks can be initiated. Then conventional programming languages are at best inappropriate as tools for thought, since they are primarily suited for expressing algorithmic processing of data. Instead we need support for expressing models of human-computer communication (and other forms of process communication) and information management in a more declarative way than the procedural paradigm of typical programming languages, i.e. descriptions oriented more towards *properties* of a system rather than explicit specifications of *how* these properties are realized.

In the area of information management this goal is pursued along several lines within different areas of research. For instance, *data abstractions* are studied in connection with programming languages, *conceptual modelling* is a vital issue within the database community and *knowledge engineering* has emerged as a subject of great practical applicability from artificial intelligence research.

The rest of this paper will be concerned with models of human-computer communication and software architecture supporting such models. The need for adequate techniques for implementation of user interfaces should be apparent from the fact that typically something like two thirds of the program text in interactive application programs are concerned with some aspect of dialogue management. Guidelines for designing dialogues and tools for their implementation have been presented in numerous papers and some books, e.g. [MAR73, FIT79, SHN80]. We will add to that tradition, but also supply some observations with general implications for the organization of interactive software, which we feel are of some importance.

The following list summarizes some properties, which we feel are important for a software system implementing a human-computer interface:

1. The internal structure of the software system should correspond as far as possible to the user's model of the system, which is expected to conform to its external behaviour. Then requests for changes in the systems behaviour are easily mapped to modifications of its internal definition.
2. The description of computations and internal data management should be separated as far as possible from the definition of

the end-user dialogue, for reasons that we be detailed below.

3. Supply of ample help information, explanation facilities and possibilities for rapid browsing of available information and operations should be encouraged.
4. Different user categories should be accommodated and systematic support for user growth, in the sense that more advanced techniques for utilizing the system is naturally acquired over time, should be part of the approach.
5. Undoing of unintended actions due to misconceptions, erroneous input or premature decisions is an important option, both for the purpose of providing safety measures and as a means to explore the behaviour of a system.

In the following sections we will discuss the consequences of such a view on the structure of software systems and on the methodology and techniques needed to develop such software. The two main issues to be treated are:

- \* The utility of *control independence* as a principle for implementation of human-computer interfaces. Under this principle we can decide on the particulars of the dialogue independently of the data processing aspects of our application. In particular *multi-style dialogues* can be supported with dynamic adaption of the dialogue to the needs of the end user.
- \* The utility of *state transition networks* for modelling the dialogue behaviour of interactive software, in the tradition reflected by the work of several others as well, e.g. [PAR69, WOO70, LUC80, DEH81].

Support for abstractions in the design of software, the notion of control independence, dialogue modelling and experiences from developing and using some tools and systems supporting the proposed approach is presented in the following sections.

## 2. ABSTRACTIONS AND DIALOGUE CONTROL.

One main trend during the history of programming languages and systems has been a striving towards higher levels of conceptual abstractions in order to promote more reliable and maintainable software systems. Another way of expressing this endeavor is as a search for an increased problem orientation in the design of applications programs, leaving details of implementation to lower levels of systems software. Programming in one sense means transforming an abstract requirements specification into a concrete executable implementation. This process is simplified, if the number of details that have to be added during the programming is confined to a minimum, i.e. if *abstractions* are supported in our programming environment.

-158-

Writing a program presumes the ability to describe a) *states* (or *objects*) b) *operations* upon states (objects) and c) *control* for sequencing operations. Traditional programming languages have a certain degree of built-in support for using abstractions when dealing with these three aspects of a program. However there is also a need for programmer defined abstractions as a means to improve software quality and productivity. Conventionally the concept of a *procedure*, performing parameterized processing of data, is the main abstraction facility. Thus by writing a procedure and specifying its input/output data relationships, a module is created which can (hopefully) be used within different environments without knowledge of its internal realization. A later generation of emerging languages, e.g. CLU [LIS77] and Alphard [SHA77], pioneered by Simula [DAH68], in addition supports the concept of user-defined *data abstractions* for the purpose of encapsulation of all information about the abstraction and thus achieving representational independence.

Data abstractions as well as the idea of schema-driven interpretation of stored representations of data in database systems, illuminates a very important concept, namely the pursuit of *data independence* in software. This is to be understood as the encapsulation of the concrete representation of data in such a way that it can be changed independently of the programs which use the data. We feel that a similar abstraction facility for the external interface to a human-computer system is very useful. The idea is to separate the description of the *contents* of a dialogue from the decision on how the actual dialogue is to be performed. Then, for instance, we can support multi-style dialogues based on either VDU forms, menu-selection, command language, etc., using the same underlying definitions of objects and operations. We will use the term *control independence* to denote such an organization of software. In section 4 of this paper examples of systems based on this principle will be given.

A variation of the same principle is when we develop a data processing application by building a *knowledge database* in the tradition of artificial intelligence research, and then implement the software as various interpreters for the same knowledge base. Then each interpreter is imposing different control and we have a control independence in the sense that the knowledge base and the set of interpreters can evolve comparatively independently.

The principle of control independence is of course implicit in much of the current work on human-computer interfaces, since techniques for table-driven screen forms dialogues, command language parser generators, etc. are in wide-spread use. However it seems to us that few systematic attempts have been made to use a common internal description as a basis for dialogues in different styles, as chosen dynamically by the end user. This matter will be further discussed in the following sections.

There are some significant short-comings associated with most existing program packages which support implementation of interactive software:



-159-

- There is often a lack of hardware independence in an implemented dialogue system. Not only may a system restrict the dialogue to a certain type of terminals (e.g. with a particular local "intelligence"), but it may also be difficult to adapt the dialogue design to new generation of terminals, changing response times or transmission speeds etc.

- Many tools are restricted in the sense that they support some specific kinds of dialogue types and exclude (definitely or almost) other dialogue organizations.

- When dialogue management is supported by adding a set of specialized macros or a library of procedures to a general purpose language, there is no guarantee that dialogue management is uniform throughout an application system.

This discussion is intended to elucidate the fact that there seems to be a shortage of dialogue design and implementation tools in the intermediate area between general-purpose programming languages extended with macro or program libraries on the one hand and special-purpose systems supporting the development of dialogue programs on the other. The approach to dialogue development presented in the next section represents an attempt to provide a comprehensive and generally useful model for the task of dialogue design and implementation. The model should be general enough to include most of the customary techniques for man-machine dialogues.

### 3. MODELLING OF DIALOGUES.

For the purpose of understanding how to design a human-computer dialogue we may assume that it is performed according to an explicit predefined *script*. The script defines which input *messages* can be understood and processed by the system. The interpretation of a message is made within a *message context*, or a *mode* as it is often called. A dialogue script contains a distinct number of such contexts and the interpretation of identical messages may differ in different contexts. In addition the *response* following a given input message also depends on the current *state* of the computation, i.e. the history of the previous interactions.

Messages accepted by the system are either *simple*, i.e. atomic tokens, or *compound*. In the latter case interactions may take place while the message is formulated. Typical examples of compound messages are screen forms or parameterized commands, while prompted responses or single command parameters may be viewed as simple messages. A dialogue script further defines the possible sequencing of messages, particularly the *transitions* between message contexts.

The concepts of message, message context and context transitions are extremely useful as a common basis for an understanding of dialogue models used in current applications software. Thus we may analyze a dialogue program from the following aspects:

- \* the number of message contexts implemented.
- \* the pattern of possible transitions between contexts.
- \* the support for compound messages.
- \* the size of the set of messages valid in a given context.
- \* the interplay between system prompts and message formulation.

Creating a description of the messages, contexts and transition patterns can be understood as defining a grammar for the dialogue. Since we have not yet discussed the *style* of the dialogue, or the dialogue technique in the sense of [MAR73], we may view the description as the *deep structure*. Various surface realizations of the dialogue are then possible and can be selected depending on the characteristics of the application or the preferences of the end user, e.g:

*Prompted input and menu-selection.* In this case there is usually a large number of contexts, one for each menu etc., and no specific notion of compound messages. Menus can be used when there is a reasonably small set of valid messages, which simplifies message formulation since numeric selection or pointing devices can be utilized. The pattern of context transitions can be arbitrarily complex.

*Command language.* Commands are usually formulated as compound messages, grouped together in a small number of contexts often arranged in a hierarchy. Since there is a finite set of commands valid within a given context, menus may be requested for selection. For each command with parameters, there is a substructure of contexts corresponding to the set of parameters. The transition between these contexts is sequential when positional parameters are used. Else there is an extra context for the set of keywords for switching to the appropriate labeled parameter context.

*VDU forms.* This case resembles the previous one with respect to the two-level structure of compound messages, although the pattern of transitions between forms is usually more varied. Within a form, each field defines a message context with transitions forced by cursor positioning on the screen. Notice however, that some transitions may be prohibited, for instance when a display-only field is not reachable with the cursor.

It should be clear from this exemplifying discussion, that the degree of user initiative in a human-computer dialogue has to do with the number of message contexts implemented and especially the permitted transition patterns. If we assume a certain *operational power* in an application system, i.e. a given set of functions that can be invoked, then in general a high degree of user initiative results from having few message contexts and user-controlled transition patterns. On the other hand more detailed messages have to be formulated in a large context, which makes such a solution less attractive. In practice, the important factor is of course how well the structure of contexts and transitions corresponds to the user's perception of the system and the way he wants to perform his tasks.

-161-

It is often assumed that command languages give a high degree of user initiative, while e.g. prompted responses gives the user limited control over the dialogue. As can be seen from the discussion above, such a statement is misleading. The style of the dialogue can usually be changed independently of the context structure.

The case where interactions between the user and the system are allowed while a compound message is formulated (e.g. correcting a validation error when entering a value for a field in a form) may be viewed as an *embedded* dialogue. Another instance of embedded dialogues may occur in systems which recognize certain inputs as *exceptional*, e.g. when interpreting a single question mark as a help request instead as a regular response. The handling of an exceptional input message is not defined locally for a context, but globally for several contexts.

#### 4. APPLICATIONS EXPERIENCES.

We have performed several application-oriented projects exploiting the approach to modelling of interactive software, which were described in the previous sections. A short description of the IDECS and MEDICS projects will be given here, as an illustration of the usefulness of the approach.

##### 4.1 The IDECS system for dialogue prototyping

The IDECS systems was developed as tool for prototyping interactive systems, where the style of the human-computer dialogue could not be decided in advance. The tool should thus support multi-style dialogues, be easy to use and allow dynamic changes of the dialogue behaviour during execution of the prototype system. The work was based on the concept of a *conversation graph*, i.e. a directed network where nodes represent message contexts and the arcs correspond to valid context transitions. The graph structure acts as a grammar for the message sequences, which may be accepted during a dialogue. The IDECS system is an interactive environment for creation and interpretation of such descriptions [HAG80]. It is implemented in Lisp [SAN78].

The conversation graph resembles very much the idea of *augmented transition networks* (ATN) in the sense of Woods [WOO70], although tokens to be parsed are rather messages (simple or compound) than single symbols. Another difference is that the conditions for accepting a message is associated with nodes and not with arcs, i.e. arc descriptions are condensed into the predecessor node. The reason for this is that interacting with a program is preferably viewed by end users as an action performed *within* a certain state, rather than as a *transition* between states. This is important for the purpose of explaining the programmed dialogue model to users and for the simplicity of the interactive tools used for dialogue description and maintenance.

Different variations of state transition diagrams have been used for the purpose of describing human-computer interfaces, e.g. [PAR69,

-162-

LUC80, DEH81] or syntax diagrams used for specification of valid language constructs in e.g. Pascal. It should be possible to support most of these approaches with the tools provided by the IDECS system.

The basic idea in the IDECS system is that an application program is organized according to the structure of the end-user dialogue. Thus each message context defines a module in the system, a *node* in the conversation graph. All information which has to do with end user interactions is represented as declarations in these nodes, and treated as named attributes. Procedural code in the system is either located to program modules, which are not allowed to interact directly with the end user, or else stored as values of attributes in the nodes.

Examples of attributes associated with nodes in the conversation graph are:

- \* *Prompts and guiding texts*, which may be dynamically selected for presentation depending on the current style of the dialogue or the preferences of the end user.
- \* *Positioning information* for prompts, to be used when a screen-oriented dialogue style is used.
- \* *Type information* for the anticipated response. This information is used to direct parsing and may also be given in the form of an invocation of a lower level conversation graph, e.g. when a compound message is to be read from the terminal.
- \* *Help information*, which can be requested as an aid for the user before the input message is formulated.
- \* *Constraints and defaults*, which should apply to entered messages.
- \* *Actions and responses*, which will be initiated when the message is processed.
- \* *Transition instructions*, which direct the transfer to another or the same node in the graph.

To write an interactive program, including the the dialogue script, in IDECS we thus have to create a set of interaction nodes and assign attributes to these nodes. For actual execution of the program a node interpreter is used, and we have also experimented with automated translations of a conversation graph to a procedural program. The correspondence between the users model of the system as a set of message contexts and the structure of the implemented program has proven very useful as an aid when the dialogue script has to be changed. Surprisingly enough experiences with IDECS show that when such a tool is used, conventional programming is almost completely eliminated for a non-trivial range of applications [HAG80].

#### 4.2 The MEDICS system for educational simulations

The MEDICS system supports interactive development, maintenance and executions of patient management problems (PMPs), to be used

for training medical students in clinical decision making. Such simulations allow the student to gather information about the patient and act in order to provide the proper management of the case. The system is of some interest for the topic of this paper, since it demonstrates how different types of control are applied to one common description of a specific PMP. These matters are further explained in [HAG82].

MEDICS is implemented with the help of a slightly modified version of the IDECS system described above. It uses the same model of the software as organized around a transition network, which defines the human-computer dialogue. However the contexts appearing as nodes in the network are more compound than in the IDECS case. Within each context the student can gather information and perform certain actions. Some actions explicitly calls for transitions to a new context. Such transitions can also be forced by the current state of the simulation. Even more than in the IDECS case, the need for explicit programming is reduced to a minimum. The fact that the program structure mirrors the structure of the dialogue makes simulation "programs" easy to understand and maintain.

## 5. CONCLUSIONS.

In this paper we have advanced the principle of *control independence* as an important tool for realizing flexible interactive software and multi-style human-computer dialogues. Our experiences indicate that many application programs are reduced to modules of limited size which implements control, acting as interpreters for application-oriented descriptive structures, when this approach is followed. The usefulness of state transition networks as a basis for dialogue software is confirmed by our experiences.

## ACKNOWLEDGEMENTS

Major contributions to the ideas and systems described in this paper have been made by Östen Oskarsson, Hans Holmgren, Olle Rosin and Roland Tibell.

## REFERENCES:

- [DAH68] Dahl, O.J., Myrhaug, B., and Nygard, K., The SIMULA 67 common base language. *Publ. No. S-2*, Norwegian Computing Center, Oslo, (1968).
- [DEH81] Dehning, W., Essig, H., and Maas, S., *The Adaption of Virtual Man-Computer Interfaces to User Requirements in Dialogs*. Springer-Verlag, 1981.
- [FIT79] Fitter, M. Towards more "natural" interactive Systems., *Int. J. of Man-Machine Studies* 11, pp 339-350, 1979.
- [HAG80] Hägglund, S., Contributions to the Development of Methods and Tools for Interactive Design of Applications Software. PhD dissertation, Linköping University, 1980.

- [HAG81] Hägglund, S., et al., Specifying Control and Data in the Design of Educational Software. *Computers & Education*, vol 6, no 1, Pergamon Press, 1982.
- [ING78] Ingalls, D.H.H., The Smalltalk-76 Programming System Design and Implementation. *Proc 5th ACM Symp. on Principles of Programming Languages*, pp 9-16, (1978).
- [LIS77] Liskow, B., Snyder, A., Atkinson, R., and Scaffert, C., Abstraction Mechanisms in CLU, *Comm. ACM* 20, 8, (Aug 1977), pp 564-576.
- [LUC80] Lucas, P., On the Structure of Application Programs, in Bjorner (ed.), *Abstract Software Specifications*, Springer-Verlag 1980.
- [MAR73] Martin, J., *Design of Man-Computer Dialogues*, Prentice-Hall, Englewood Cliffs, New Jersey, (1973).
- [PAR76] Parnas, D. L., On the Design and Development of Program Families. *IEEE Trans. on Software Eng. SE-2*, pp 1-9, (Mar 1976).
- [SAN78] Sandewall, E., Programming in an Interactive Environment: the Lisp Experience. *ACM Comp. Surveys*, vol. 10, no 1, pp 35 -71 (1978).
- [SHA77] Shaw, M., and Wulf, W.A., Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators., *Comm. ACM* 20, 8, (Aug 1977), pp 553-564.
- [SHN80] Shneiderman, B., *Software Psychology*, Whintrop Publishers, Cambridge, Mass., (1980).
- [WOO70] Woods, W.A., Transition Network Grammars for Natural Language Analysis. *Comm. ACM*, vol 13, no 10, pp 591 - 606 (1970).

Displays of Program Structure: Why and How?

T.R.G. Green

MRC/SSRC Social & Applied Psychology Unit

Dept of Psychology

The University

Sheffield S10 2TN

1. General outline

To comprehend, debug, modify, update, document, or even to recognise a program, information must be extracted from the written notation. However, some types of information are hidden in the text, rather than manifest, which means that the programmer has to go through a series of mental operations. ("Where can this subroutine be called from?", for instance, usually requires a scan of a good deal of text). There are notational devices which will help to improve the visibility of information, and there are structural devices which help to reduce the complexity of information, but it is hard to see how all the information that programmers need can easily be made visible at the same time.

The written notation in its usual form is also very bulky and very ill-cued. Unlike normal printed English it contains neither typographical cues, which in printed English can be both diverse and ingenious, nor verbal cues such as "Next, ..." or "In contrast, ..." and the like. Some have diagnosed this as solely a problem of bulk, and have invented very compact notations such as APL. Others have tried using a perceptual coding rather than a symbolic one, and have developed flowcharts and structure diagrams. Recently there have been attempts to create abbreviated displays, by replacing the inner loops of programs with a row of dots. Unfortunately, this approach sometimes hides the very information the programmer is seeking.

There are existing algorithms capable of revealing much of the information needed by the programmer, and there are possible approaches to abbreviating the display of information in more effective ways. This paper outlines a proposed project to evaluate their practical utility. Specifically, we wish to take a real-life

Thomas Green

class of programs and provide an interactive tool for analysing them.

A suitable class of real-life programs seems to be Basic programs for hobbyists, especially games. These are short but interestingly filled with logic, and their typical dirty style will be a good test. Working with Basic programs will help to concentrate our minds on the need to provide an interface of extreme utility, since the typical hobbyist will lose patience quickly with anything which demands unnecessary effort. There is nothing very sacrosanct about the choice of Basic - it merely happens to fit the bill; another interesting choice would have been Cobol.

The task of a tool to assist comprehension of these programs would be:

1. Provide a mechanism for answering certain common types of question, such as "How can this program reach this point?", "What does the value of this identifier depend on at this point?", etc. These questions can be answered mechanically with no specific knowledge of programs and their components, but the degree of detail to present will need some careful thought.
2. Provide a mechanism for recognising the simpler components of programming. There is no challenge in recognising those components that are defined by control structures, such as for-loops and procedures; these are easy to recognise by algorithm but just as easy for the reader to spot (in Pascal, at least; Basic is another story). The task is recognising 'schemas', bringing together statements which may be quite far apart from each other. The use of a flag or state variable is a good example. Clearly this is a step towards a knowledge-based expert system, but not a very large step as presently envisaged.
3. Present abbreviated displays. The ideal abbreviation would suppress all material that is unimportant to the task in hand, and would cue the remaining material according to importance.

Each of these points will be discussed in more detail below.

## 2. Background: Sheffield



Thomas Green

Our team at Sheffield has conducted a number of investigations of the effects of programming language design on programming performance. Two are particularly worth noticing here. In the first, Sime, Green and Guest (1977) compared the performance of novices writing simple conditional programs in three different languages. Nested conditionals, in the Algol tradition, produced significantly fewer errors of logic than jump-style conditionals, in the Fortran tradition. But the 'error lifetimes' - the number of debugging runs required, given the first run failed - were very similar for Algol-like and Fortran-like conditionals. A third language, however, called Nest-INE, which used the same logical structure as Algol but which was designed to facilitate answering questions, brought the error lifetimes down to one-tenth of their Algol/Fortran-like values. The second experiment was designed to extend this result, by investigating professionals instead of novices, in a comprehension exercise instead of a programming task. In it, Green (1977) showed that questions of the form "Under what conditions can this program do such-and-such?" were answered significantly faster in Nest-INE than in the Algol/Fortran-like dialects.

The difference between the Algol-like dialect and Nest-INE is very small. In the Algol-like style, conditionals used the conventional else and were not explicitly ended. Nest-INE was more explicit:

```
IF predicate: . . . .  
NOT predicate: . . . .  
END predicate
```

These two experiments show first, that a small change in the language can profoundly affect performance, at least for novices; second, that it can affect professionals as well, though not to the same degree; and third, that the effect on novices appears to be caused by changes in the ease of extracting information from programs, rather than by changes in 'writability'.

These experiments were performed with extremely small programs. It is a reasonable hypothesis that the difficulty of extracting information rises very sharply as programs get longer. In real programs, programmers will need a variety of different types of information, about data flow as well as control flow; but the overall principle should still apply.

### 3. Some example techniques

#### Dependency analyses.

The methods of static analysis (Cocke and Allen, 1976) are now well-known. The program is divided into chunks or 'intervals' which are entered only from the head. The interval-graph can then itself be analysed to produce higher-order intervals, and so on until an irreducible graph is obtained. In this process it is not difficult to specify the data dependencies between chunks, showing which variables are live at a given point, etc.

From such a graph it is also possible to work backwards from a given point to record the paths which enter a given chunk. Thus it is easy to present at least some information in answer to the question "How can this point be reached?" It will be a problem to present it digestibly, but according to the results of our experiments it is vital to do so.

#### Components

The perception of program structure clearly demands labelling the uses to which chunks and variables are put. Given the data dependency analysis, certain types of chunk are quite easy to label: straightforward FOR-NEXT constructions fall out of the syntax analysis, of course, but while-type loops can also be identified readily enough, if their construction follows standard forms. Simple subroutines where all the "COMESUB" statements precede the RETURNS are also quite straightforward. Simple IF-THEN-ELSEs, even nested, should be identifiable, as long as the branches re-unite at some point.

Possible answers to "What does variable V do?" (in the world of hobby programs) include: state variable; simple counter; running total; arithmetic computations; transmitter of parameters and results; a constant (set only once); and of course Don't Know. Each of these can be identified in straightforward cases.

#### Abbreviations

It is not easy to decide on an effective strategy for abbreviations. Consider the following fragment:

```
total := 0;  
for j := 1 to N do
```

```
        if A[j] > 0 then total := total + 1 ;  
writeln (total)
```

What we do not want to do is to abbreviate that into

```
total := 0;  
for .....  
writeln (total)
```

Yet that is exactly what many structure-based editors do (eg the Cornell Program Synthesizer).

One approach might be to present a display that was organised by degree of importance. This approach comes from work on discourse analysis, notably Kintsch and van Dijk (1978). Unfortunately it is a characteristic of their method of analysis that the 'importance' of a proposition depends on the number of other propositions for which it is a precondition of interpretation; and taken literally, in a program the final output statement tends to depend on the entire rest of the program! It is not clear at the time of writing whether a satisfactory solution can be found.

### Conclusions

The three steps of the previous section display an increasing departure from algorithmic methods and an increasing reliance on identifying forms of analysis which are psychologically useful. At one end, mechanical labour-saving techniques are well-understood, and our previous experiments indicate that they will be very useful if well interfaced; at the other end, the possibility of an importance-analysis, rather than presenting a program abbreviated by control-depth, is anything but well-understood.

At the time of writing developments are being made in the first version of such a tool. Examples of the the type of analysis it generates will be given and the problems of an interface will be discussed. From this point we intend to perform experiments using simulated software, to determine the efficacy of various options, before evaluating a working version.

### References

Thomas Green

Cocke, J. and Allen, F. E. (1976) A program data flow analysis procedure. Communications of the ACM.

Green, T. R. G. (1977) Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 50, 93-109.

Kintsch, W. and van Dijk, T. A. (1978) Toward a model of text comprehension and production. Psychological Review, 85, 363-394.

Sime, M. E., Green, T. R. G., and Guest, D. J. (1977) Scope marking in computer conditionals - psychological evaluation. International Journal of Man-Machine Studies, 9, 107-118.

The Evaluation of a Programming Support Environment

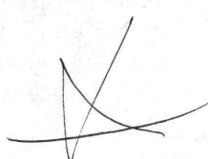
---

Andrew Arblaster.  
Queen Mary College  
University of London

## INTRODUCTION

This report is an account of a study carried out by Logica Ltd. for the UK Royal Signals and Radar Establishment.

The emergence of Ada as a standard programming language will have a major impact on the quality of embedded computer systems. This impact will depend as much on the quality of Ada Programming Support Environments (Apses) as on the language itself. A great influence on the quality of an Apse is the Human Factors quality of that Apse. The purpose of the project reported here was to evaluate the design of one particular Apse, produced for the UK Department of Industry and Ministry of Defence and known therefore as the 'UK Apse' (DoI, 1981). This report does not discuss the details of the design of the UK Apse or the detailed recommendations made as a result of the study. We concentrate here on the methods used to evaluate the design and the relative effectiveness of these.



The factors included in the evaluation were the external user interfaces, the sort of things usually meant by 'Human Engineering', and also issues related to the structure of the Apse. In the first category are such things as command language features and in the second the functions of components of the Apse and the interactions between them. In addition it was necessary to consider issues related to the use of the Apse as an organisational component in software projects - its role in software management besides its role as a set of software tools. When we consider all of these factors we can see that the term 'programming support environment' is not entirely apt. 'Programming', unless we radically widen its definition, is very far from the whole story where software development is concerned. Consideration of existing software tools in isolation and of structural requirements for existing environments demonstrates that support is needed, and is often offered, for a much wider range of activities than what is normally understood by 'programming'. A support environment intended to host development of large scale systems must provide support for activities throughout the software lifecycle. Most of these activities are influenced greatly by human factors.

The study itself was conducted in three phases. First a survey of the current state of knowledge of human factors as applied to programming support environments was conducted. Secondly the STONEMAN requirements for Apses were studied with the aim of determining the human factors implications of these requirements. The third phase of the study was the evaluation proper, concentrating on the human interfaces and on the structural

features of the UK Apse design. The <sup>172</sup>last phase embodied two 'rapid prototyping' exercises; one studied the implementation and use of the command language interpreter, and the other a particular application of the Apse Database. The study was therefore a survey of 'applicable' knowledge and an application of that knowledge to evaluation of a very substantial software system at the design stage. The applied part of the study was focussed specifically on aspects of UK Apse design, though it is hoped that the evaluation methods used can be applied to other software systems and that the findings will have implications for other Apse designs. This form of 'applied' project in the human factors field is very much the exception rather than the rule: in this field the aim of most empirical studies is to develop 'applicable' design principles, rather than to apply those principles to a particular design. Of course, much anecdotal 'work' also exists - armchair musings which may have some value but which can provide only the most fragile underpinnings for evaluation of a software system design.

#### THE SURVEY

The survey phase of the study outlined the factors to be considered from the point of view of the comparatively new field of software engineering, then from the point of view of human factors work, after which a synthesis was attempted. The design of programming environments was then considered within the framework which had been built up. Finally critical elements of the framework was incorporated in two checklists, the first relating a software engineering taxonomy of software quality (figure 1) to human factors issues and the second a consolidation and abstraction of different checklists which have been proposed for interactive systems design criteria. These checklists were developed for use in later phases of the study as evaluation guidelines.

#### STONEMAN REVIEW

The basic requirements document for ApSES is STONEMAN (DoD 1980). The second phase of the study investigated the congruencies between human factors issues and the STONEMAN requirements. During the evolution of the Apse concept since STONEMAN considerable development and modification of the ideas which had been put forward proved to have taken place; however, it was found that the STONEMAN requirements were generally not inconsistent with high human factors quality. *m*

#### EVALUATION

The evaluation phase itself was intended to identify :

- fundamental aspects of the design which needed to be changed for human factors reasons. It was hoped that these would be few.
- more superficial improvements which would nevertheless have a significant impact on human

factors.

- additional software tools and capabilities required to make the UK Apse understandable, usable and robust.

This report does not describe the detailed recommendations of this phase of the study, but concentrates on the methods used to arrive at those recommendations.

Particular empirical investigations supporting the evaluation were:

- construction of, and experimentation with, a prototype model of the Command Language interpreter.
- the construction of a prototype Kernel Apse database model. This was an ad hoc realisation of a particular database fragment together with a limited prototype Database Utility. The database fragment used was taken from a data model of a part of a software engineering system which was being developed at the same time as the study.
- The development of a scenario of use of the Apse for a nontrivial test case project which was assumed to be being developed using the UK Apse facilities. This was intended to provide a focus for team discussions of situations facing a typical software development team using the Apse. The objective was to provide a starting point for consideration of likely patterns of use of the Apse. The scenario also provided ideas of likely commands which would be issued to the Command Language Interpreter and a data model for trial using the database prototype.

## CONCLUSION

Finally this report comments on the success of the methods of evaluation used in the study, and the lessons for future activities of this sort. The prototyping exercises were particularly successful in the case of the Command Language Interpreter, where it was felt that much of value was learned during the implementation, even before trials of the system.

Broadly, factors related to perceptual properties, cognitive properties and available feedback were identified as being critical. Different empirical activities provided information on different of these factors.

The relationship of these psychological factors to software quality factors will be described.

## REFERENCES

- DoD 1980: Requirements for Ada Programming Support Environments: "STONEMAN". US Department of Defense.
- DoI 1981: UK Ada Study Final Technical Report. UK Department of Industry.

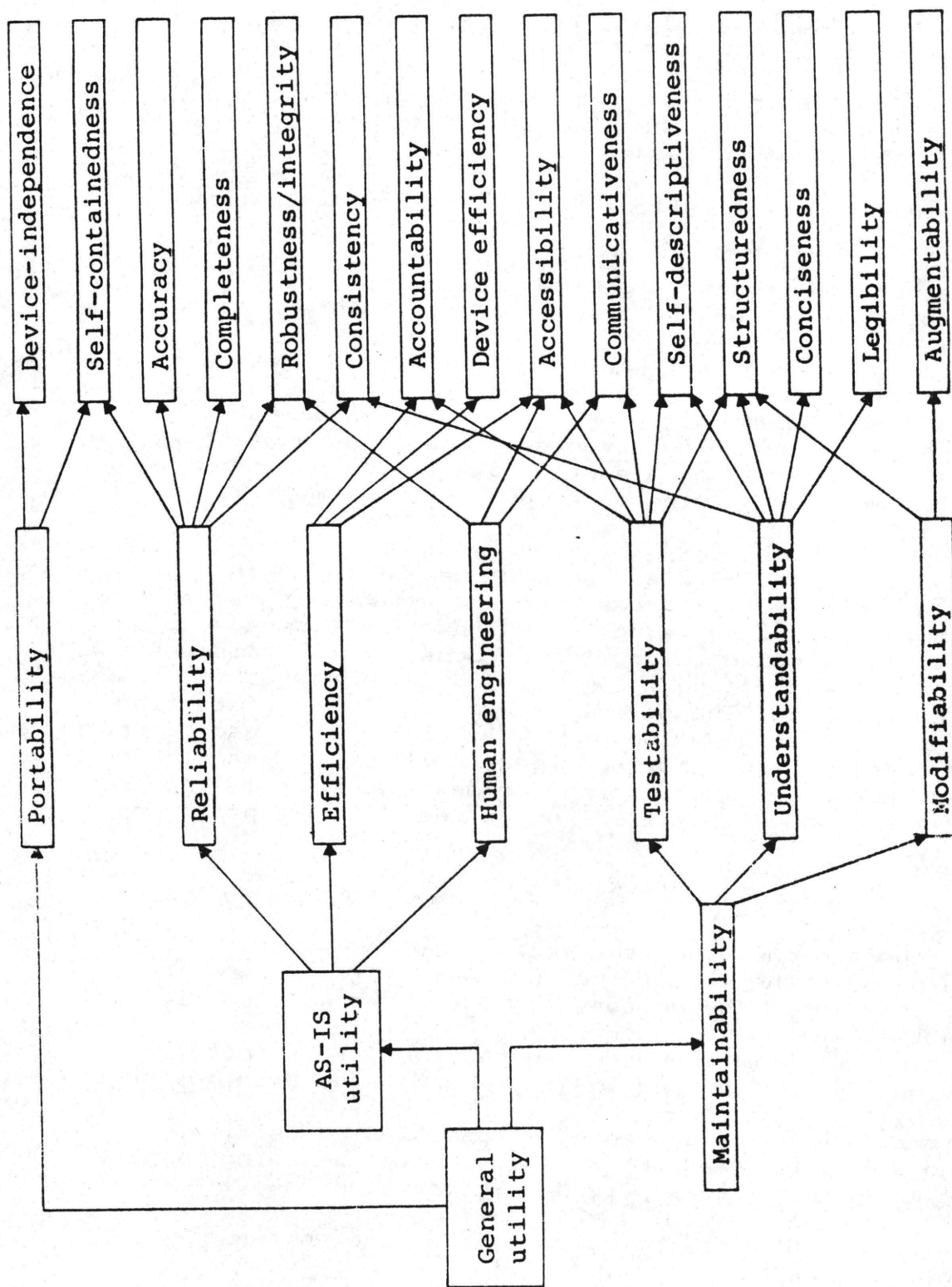


Figure 1: Software Quality Characteristics Tree



II. FACILITATING HUMAN-COMPUTER INTERACTION

B. System design issues



## Why Systems Transparency ?

=====

Susanne Maass  
Fachbereich Informatik, Universitaet Hamburg

Userfriendliness, ergonomic systems design, systems flexibility, ease of use, systems transparency - these are some of the increasingly well-known catchwords in computer human factors discussions of the last five years (cf. DEHNING/ESSIG/MAASS 81). This paper will concentrate on the issue of systems transparency. It characterises the computer not any more as a number crunching machine but as a communication medium, even as a virtual communication partner programmed for the performance of certain roles.

The analogy with human communication provides concepts for a new understanding of the users' situation in their interaction with a computer system. It will become obvious that systems transparency should be a central issue in the discussion of human factors and userfriendly systems design.

Systems transparency can be defined in different ways.

- 1- A transparent system does not hide any of its functions and mechanisms from the user (you can "look inside").
- 2- A transparent system does not obscure the user's view on the problems he wants to solve with the computer (comparable to a well cleaned window).
- 3- A transparent system appears well structured, consistent and comprehensible to its users; the users can easily build up an internal model of the relevant parts of the system.
- 4- Transparency is the degree to which the logical user interface conforms with the user's prior knowledge or human intuition. (Notice the reference to the individual user!)

Our later definition will integrate 2, 3 and 4 .

### Work with computers

-----

Computers are said to be universal machines which can be specialised by programming. A specific characteristic of these machines is that we manipulate them by language: By means of words we indicate the functions to be performed and (in most cases) we see the effects that have been achieved displayed verbally on the screen. Especially important is

the fact that the computer can comment its own functioning; it can tell the user how to handle it in what situations for what purposes. So the notions "human-computer dialog" and "human-computer communication" have come up to characterise the resulting particular system - user relationship.

But how do people work with computers ?

Let's take the example of office computers.

In most cases computers - as other technologies before them - are being introduced for rationalisation purposes (cf. BRIEFS 80). The permanent standardisation and formalisation of work processes in the past and the reorganisation of work by increased division of labour have prepared the ground for the advent of computers.

Instead of using documents, reports, filing cards and forms the office worker now sits at his VDU screen entering data, following programmed procedures and watching the results. For this he has to learn how to handle the system, i.e., the dialog commands and data formats. A great part of the experience he had with the office procedures he performed before is not important any more. The procedures are incorporated in the programs he's now working with.

The system's behaviour and by it the surrounding work organisation is thus determined by the systems designers and programmers. Users normally only get some minimal information about the system. In many companies it is common to give them a short introduction in a training session of 2 - 3 hours at the terminal. After that they certainly do not understand how it works or how to recover from problematic situations in dialog. Their own role has been defined without them being able to take too much influence.

This short description of people's working conditions in the computerised office should have made clear that in many applications users do not perceive the computer just as a handy tool. (They possibly rather get the feeling of themselves being tools for running the machine.)

And how could they - additionally considering the fact that the system may permanently monitor their performance data at the terminal: their reaction times, their mistakes, their preferred function sequences, their break times ... as common management information systems do.

In our opinion the situation could be partially improved by transparent systems design, which helps the user to understand the machine and make use of it. Of course the indicated control effect cannot be removed by systems transparency as it will be defined later on.

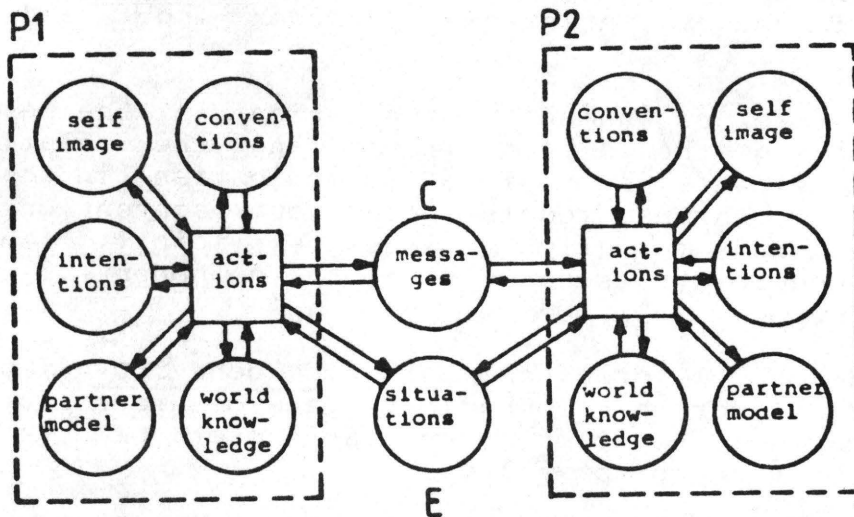
We'll compare human interaction with computers to an extremely restricted kind of communication, namely that of formal or even algorithmic communication. This analogy will explain the vital importance of systems transparency.

Natural versus formal communication

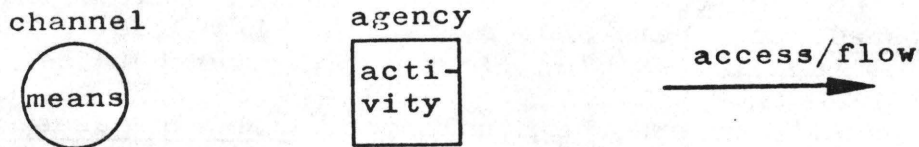
The study of psycho-linguistic literature (see e.g. LAING et al. 66, LEWIS 69, GLOY/PRESCH 75) has lead us to a rough model of natural communication. For representation we use channel/ agency and means/activity nets, which are special kinds of Petri nets (see OBERQUELLE 80).

Communication is seen as a complex of social actions for the purpose of understanding and of allowing coordinated actions. It takes place by means of a medium, the communication channel (C), and involves at least two communicating agents (P1 and P2). In the model we seperate the communication medium from the rest of reality (the environment E) which surrounds the partners.

Several factors are supposed to influence people's communicating behaviour. \*



A model of communication  
(channel/agency and means/activity net representations combined as indicated below)



\* Of course this kind of formal model cannot deal with all aspects of the complicated phenomenon of communication. However, it already suffices to provide new insights.

In communication the sender wants to achieve some effect in the recipient, i.e., he has some underlying intentions which guide the planning process for his messages.

In order to be comprehensible for the partner he orients along certain common conventions and a partner model. In natural communication social norms of conduct and social roles are reflected in the applied syntactic, semantic and pragmatic linguistic conventions. Partner models help to predict the partner's behaviour and expectations.

The statement planning process is also influenced by the sender's self image, i.e., his self confidence, his conscious habits, the role he considers himself to be in... Apart from knowledge about conventions the sender uses his world knowledge.

The recipient tries to reconstruct the planning process of the message to understand it. To comprehend its meaning he has to find out the factors that influenced the partner in formulating his statement. This understanding process and the subsequent actions of the recipient of the message in turn depend on his set of conventions, self image, intentions, partner model and world knowledge.

An essential feature of human communication is the permanent possibility of making the dialog itself and the relationship between the participants a topic of communication (in so-called "metacommunication"). Partner models, intentions, conventions etc. can be discussed and mutually modified explicitly. (This is indicated by the double arrows between all the components.)

An agent is said to show formal communicating behaviour if all the components relevant for his behaviour can be described by mathematical models. Sometimes just a subset of the model components might be relevant at all.

Formal communication takes place if at least one of the participants shows formal communicating behaviour. By this he indirectly forces the other participants to adapt themselves and communicate formally as well, since if they didn't they would risk misunderstanding.

In many cases formal communicating behaviour of people can be explained by the fact that they act in certain roles which have been delegated to them and by which they feel restricted: Think, e.g., of an inflexible clerk. (For further details see KUPKA/MAASS/OBERQUELLE 81.)

Delegation usually covers the handling of standard cases only. For exceptional cases it must be clear who identifies them as such and what has to be done. This is no problem in human communication, where people play roles consciously and are able to leave their role temporarily to cope with unexpected situations.

## Human - computer communication

---

The programming of computer systems can be considered as an extreme form of delegation. Here the relation between behaviour and its preconditions is not just formally but even algorithmically described. The delegation of tasks to computers requires the previous formalisation of the delegated work processes.

Designers and programmers of interactive systems do not only delegate functional behaviour (problem solving functions) but also communicating behaviour (how to handle the virtual problem solving machine by means of dialog functions). They design virtual communication partners for the users. This is why we propose a new paradigm which characterises computers as communication media with formal communicating behaviour (see KUPKA/MAASS/OBERQUELLE 81 and MAASS/OBERQUELLE/KUPKA 82).

In the normal systems design process the designers assume certain user characteristics, i.e., they have certain - not extremely realistic - user models in their minds which get formalised and implemented by dialog conventions.

Later, the naive non-programming user has to get on with these restricted communication conditions. In his dialog with the interactive system he must adapt to the formal models which do not necessarily match his real needs and expectations and must follow the conventions.

Only those users who are able to re-program the machine can theoretically modify the implemented user model to their own liking. In practice, however, this kind of machine adaptation will hardly ever be done because of the complexity of computer systems.

We conclude:

Computer systems are extremely inflexible virtual communication partners. They force the human user to adapt to a strictly formal communicating behaviour. For this purpose the user has to build up a suitable system model in his mind which helps to predict and explain the system's behaviour.

## Systems transparency

---

In our opinion transparent systems design can help the user in this situation of compulsory adaptation.

We define transparency as follows:

A transparent system makes it easy for users to build up an internal model of the functions the system can perform for them. This includes the problem solving functions as well as the dialog functions.

Its logical interface conforms with the users' prior knowledge about the problem domain and with human intuition (i.e., how they are used to coping with certain situations in communication). The effort of handling the system (by dialog functions) must not disturb the users' problem solving processes.

This definition is confined to the users' immediate tasks. A user understands the virtual machine he's working with, but he does not necessarily see through the other functions it has, for instance, for his manager or for the personnel division of his company.

So, what makes a system transparent to its users ?

Obviously, a system which is transparent to one user need not necessarily be transparent to others as well. The ability of understanding the system depends on the user's knowledge and experience. (We can thus make a given system transparent even without modifying it - just by an extensive information and training of its users.)

However, referring back to the components of our communication model we can derive some general guidelines for transparent design of human-computer communication.

Dialog conventions (syntactic, semantic and pragmatic) should appear as natural as possible to the user.

- \* For problem solving functions use the language of the application field or technical terms that are common to the user.
- \* Allow abbreviations for all kinds of functions.
- \* Avoid artificial expressions that are "natural" only to the designers (e.g. in error messages, but also for the required commands).
- \* Allow underdetermined commands (incomplete parameter specification) and let the system ask back.
- \* Allow metacommunication, e.g. questions for available functions and obligatory formats, choice of different interaction modes, modification of defaults, definition of complex commands (procedures).
- \* In particular, allow questions at any time.

A special problem is caused by the fact that the user is rarely confronted with one consistent interface: Computer systems consist of a number of different programs - like operating system, editors, compilers, application packages etc.. These subsystems have been designed independently by different people who have each implemented their own rather individual conventions.

- \* Provide a consistent user interface: similar user commands for similar functions, consistent conventions concerning abbreviations etc., consistent reactions to user mistakes...



\* Make the system "have a self image" in that it is able to explain its behaviour, its conventions, its user model..

Apart from systems transparency there are also other requirements for better systems design that can be explained by the presented concept of formal human-computer communication. For example, the need for participative systems design, or even "users' design" (cf. EASON/DAMODARAN 81): In our terminology it means that the users get the opportunity of designing the system's user model themselves, and of making the system use their own conventions.

User participation is started being realised by now and if it is practised the right way it has undeniable advantages for the users. But there are also serious fundamental disadvantages in that systems designers profit from the worker's job experience to computerise work processes as far as possible - with the effect that the worker's qualification may largely become superfluous and with it the whole job.

Obviously, userfriendly design and user involvement alone are not sufficient to guarantee humane jobs and cannot reduce the function of computer systems to being simple tools for its users. Too significantly does their introduction affect the working conditions of the persons concerned.

Literature

BRIEFS 80

Ulrich Briefs:  
"Arbeiten ohne Sinn und Perspektive ?  
Gewerkschaften und 'Neue Technologien'"  
Pahl-Rugenstein: Koeln, 1980.

DEHNING/ESSIG/MAASS 81

Waltraud Dehning, Heidrun Essig, Susanne  
Maass:  
"The Adaptation of Virtual Man-Computer  
Interfaces to User Requirements in Dialogs"  
Lecture Notes in Computer Science no 110,  
Springer: Berlin, Heidelberg, New York, 1981.

EASON/DAMODARAN 81

Ken D. Eason, Leela Damodaran:  
"Design Procedures For User Involvement  
and User Support",  
in: M.J.Coombs, J.L.Alty (Eds.):  
"Computing Skills and the User Interface"  
London: Academic Press, 1981  
pp. 373-388.

GLOY/PRESCH 75

Klaus Gloy, Gunter Presch:  
"Sprachnormen",  
Vol. 1-3,  
Frommann-Holzboog: Stuttgart, 1975.

KUPKA/MAASS/  
OBERQUELLE 81

Ingbert Kupka, Susanne Maass, Horst Ober-  
quelle:  
"Kommunikation - ein Grundbegriff fuer die  
Informatik",  
Mitteilung Nr. 91, IFI-HH-M-91/81, 1981,  
Universitaet Hamburg, Fachbereich Informatik.

LAING/PHILLIPSON/  
LEE 66

R.D.Laing, H.Phillipson, A.R.Lee:  
"Interpersonal Perception",  
Tavistock: London, New York, 1966.

LEWIS 69

David Lewis:  
"Convention - A Philosophical Study",  
Cambridge/Mass., 1969.

MAASS/OBERQUELLE/  
KUPKA 82

Susanne Maass, Horst Oberquelle,  
Ingbert Kupka:  
"Human-Computer Communication :  
Towards a New Understanding",  
in: Najah Naffah (Ed.):  
"Office Information Systems",  
North-Holland: Amsterdam, 1982,  
pp.551-561.

OBERQUELLE 80

Horst Oberquelle:  
"Nets As a Tool in Teaching and Terminology  
Work",  
in: Wilfried Brauer (Ed.):  
"Net Theory and Applications",  
Lecture Notes in Computer Science Vol. 84  
Springer: Berlin, Heidelberg, New York, 1980,  
pp.481-506.

Léonardo Pinsky \*

1. Introduction

What is one to call the interaction between a computer and its operator ? Considering the range of tasks and the wide assortment of users that perform them , the answer to this question is not immediately apparent. Nevertheless " Dialogue ", evoking a characteristically human mode of communication, discourse between two people, is very widespread. Is it a useful or a confusing term ?

From the prescriptive point of view Smith (1980) has urged many, serious objections to the argument that natural language ought to be the model for the design of interactive systems. Nevertheless, (a) some systems have to deal with information in a natural language , and (b) to enable operators with little or no formal training to use computers, interaction facilities must be very similar to a natural language.

One may also wonder from the descriptive point of view how adequate "dialogue" is for the actual interactions between users and computers ? I will consider this question in the context of a particular working situation, data entry and coding on line.

The research described below was conducted in two stages: (1) an analysis of operators work using a first system (Pinsky et. al. 1979), and (2) experimentations with a group of operators during the design of a new system .

\* Laboratoire de Physiologie du Travail et d'Ergonomie.  
Conservatoire National des Arts et Métiers, Paris, FRANCE.

## 2. Principles of the data entry-coding system

Information collected in an investigation has to be coded in order to build a data base for further statistical operations.

People had filled out a printed form (fig. 1) containing "closed" questions (like 15 in fig. 1) and "open" ones (like 12 and 14) concerning their profession, the name, economic activity, and address of the firm where they work, and so forth.

**POUR VOUS REMPLIR CE QUESTIONNAIRE**

• Vous exercez une activité professionnelle : répondez aux questions 12 à 15.  
 y compris si vous êtes un membre de votre famille dans son travail, même à temps partiel ;  
 si vous êtes apprenti, sous contrat ou stagiaire rémunéré ;

• Vous n'exercez pas actuellement d'activité professionnelle ou vous êtes au chômage : répondez à la question 16.

**12** Indiquez la profession ou le métier que vous exercez actuellement.  
 Répondez par exemple : ouvrier électronique d'entretien chauffeur de poids lourds, directeur d'école ou électricien, ingénieur de maintenance, vendeur en électroménager, employé de comptabilité, etc.  
 Adressez-vous un membre de votre famille dans son travail ? (Établissement agricole, artisanal, commerce, profession libérale, etc.) OUI  NON

**13** Exercez-vous cette profession comme :  
 • Employeur ou travailleur indépendant (chef d'entreprise agricole ou consultant, artisan, commerçant, industriel, membre d'une profession libérale, etc.)  1-  
 • Aide familial non salarié (compagnon, enfant ou autre membre de la famille d'un agriculteur, d'un commerçant, etc.)  2-  
 • Apprenti sous contrat  3-  
 • Salarié  4-  
 Employeur-est-ce des salariés ? OUI  NON   
 Si oui, combien ? (1 ou 2)  1, (3 à 9)  2, (10 ou plus)  3  
 Dans l'agriculture, indiquez également les salariés permanents. NON  OUI   
 Êtes-vous travailleur à domicile pour le compte d'une ou plusieurs entreprises ? OUI  NON

**14** OU TRAVAILLEZ-VOUS ?  
 ADRESSE de votre lieu de travail  
 N° 18 Rue (ou hauteur) rue Davout  
 Commune et dépt. 21000 Dijon  
 (Pour Paris, Lyon, Marseille, précisez l'arrondissement)  
 NOM (ou raison sociale) de l'établissement (industriel, commercial, administratif agricole, etc.) qui vous emploie ou que vous dirigez : ERCA  
 (Pour Paris, Lyon, Marseille, précisez l'arrondissement)  
 ACTIVITÉ de cet établissement :  
 Répondez par exemple : commerce de gros, fabrication de vêtements, travaux métalliques, filage de soie, transport routier de voyageurs, etc.  
 Adresse de cet établissement, si elle est différente de celle indiquée à la question 14 a :  
 N° Rue (ou hauteur)  
 Commune et dépt.  
 (Pour Paris, Lyon, Marseille, précisez l'arrondissement)

**15** POUR LES SALARIÉS  
 Indiquez la catégorie professionnelle de votre emploi actuel :  
 • Manœuvre ou manœuvre subalterne  1  
 • Ouvrier  2  
 • Ouvrier qualifié (O.E. O1, O2, O3)  3  
 • Ouvrier qualifié (P1, P2, P3, TA, OP, OQ)  4  
 • Employé  5  
 • Technicien, dessinateur  6  
 • Agent de direction des ouvriers ou des employés  7  
 • Ingénieur ou cadre (sauf employés, techniciens, agents de maîtrise et agent pour le contrôle de cadre ou directeur par intérim ou directeur d'école)  8  
 • Autres cas  9  
 Si vous êtes agent de l'État, d'une collectivité locale ou d'un service public (D.T., S.N.C.F. ou service de santé, précisez votre grade. Exemple : contrôleur du Trésor, receveur P.T.T. de 4<sup>e</sup> classe, agent des services postaux).  
 Si vous êtes ingénieur, cadre, agent de maîtrise ou technicien, précisez votre fonction principale dans l'entreprise ou l'organisme où vous employez :  
 • Directeur général ou un de ses adjoints directs  1  
 • Fonction administrative, financière ou comptable  2  
 • Production, fabrication, chantiers  3  
 • Entretien, travaux publics, maintenance, dépannage  4  
 • Etudes, essais, métrologie, recherche  5  
 • Autres fonctions  6  
 Précisez (informatique, sécurité, santé, ...)

Figure 1

The operator has to enter the precoded answers and to code those in ordinary language using the interactive system. The interaction style is to fill in the blanks of a display form on the screen (fig. 2), to transmit the information to the computer, and to wait for responses.

display form  
 →  
 responses of the computer  
 →

```

0221 SUI FOR MODE 0 MAJ RECH
ECH 20 -D- 59 -C- 178 DBT AM03 IMM 001 LOG 04 CP- MO- 0009 TA- 1
PROFESSION DIRECTEUR ADJOINT 12-87 16
14A-ADRESSE LT:MO- 10 14A-RUE R DAYVOUT
14A-COM DIJON 14A-DEP 21
14B-RS CAISSE REGIONALE CREDIT AGRICOLE
14C-AE BANQUE
14D-ADRESSE ET:NO- 14D-RUE
14D-COM 14D-DEP
15A-CPF 0 15C-FDNC 1 P75 9916 SAU OPA PUB -7-

01 CAISSE REGION CREDIT AGRICOLE 010 R DAYVOUT 8903 03
02 CAISSE REGION CREDIT AGRICOLE 110 AV EIFFEL 8903 05
03 CAISSE REGION CREDIT AGRICOLE 00 BACHELAR 8903 03
04 CAISSE REGION CREDIT AGRICOLE 103 AV DRAPEAU 8903 04
05 CAISSE REGION CREDIT AGRICOLE 089 AV HUGO 8903 06
06 CAISSE REGION CREDIT AGRICOLE 013 PL DARCY 8903 12
07 CAISSE REGION CREDIT AGRICOLE R JOLY 8903 07
08 CAISSE REGION CREDIT AGRICOLE 004 PL BANQUE 8903 15
09 CAISSE REGION CREDIT AGRICOLE 060 R AUXONNE 8903 06
10 COMMISSI REGIONAL AGRICOLE 005 R RENAUD 9102 02
    
```

Figure 2

The declarations entered by the operator are treated sequentially by the computer:

- it searches for the firm in a catalogue, if it's found, the economic activity of the firm, its legal status, size and address are automatically coded. Otherwise,
- it searches for the firm's economic activity as entered by the operator in a file of pre-established designations (about 4,000). If it's found, it is coded; if not, either lists of designations containing recognized words or messages in a natural language are sent back to the operator.
- In the same way, the computer searches for the firm's address.
- For the profession, the computer uses the description entered by the operator as well as other pieces of information (e.g., economic activity, wage-earning status, professional category, duties, size of the firm...).

It searches first in a file of pre-established designations (about 9,000); but unlike the situation with respect to economic activities, the correspondence between the description entered by the operator and the designation in the file need not be word-for-word. The designation REPAIRMAN for example, is valid for all job titles beginning with this word (REPAIRMAN, TELEVISION; REPAIRMAN, WASHING MACHINE; REPAIRMAN, ELECTRICAL, etc.). The other pieces of information are used as variables for decision tables. Which ones will be employed in a given case depends upon the profession in question. If a designation is found and if the algorithm of the decision tables succeeds, a code is assigned; otherwise, messages are sent back to the operator. The codes for economic activities and professions correspond to the categories that subdivide the designation files.

If a code is assigned a brief description of the category's contents is sent back to the operator for his approval.

3. Interaction analysis from the operator's point of view

The analysis is based on two types of record:

- observable behavior: video tapes were made of the operator's actions (filling in or modifying the display form), his eye movements (to determine when he was looking at the screen, the display form, or the keyboard), and the computer's responses.

-verbal protocol :tape recordings were made of the operator's remarks both while executing the task and afterward.

The interpretation of the data so collected requires extensive discussion with the operators. Close cooperation with them, then, is an essential feature of the method.

In what follows, I seek only to sketch a framework for the analysis. By examining the circumstances in which something goes wrong during the interaction, we may hope to learn more about the interaction and the operator's cognitive activities.

It often happens that the operator does not understand the computer's response or thinks it is improper or even is lost . This is especially evident when considering the verbal protocol. Let us give a few examples.

(A) Operator: profession= Doctor of Urology  
economic activity= Hospital

Computer: DOCTOR OF MEDICINE..., DOCTOR... (WITHOUT A PRIVATE PRACTICE)  
-SPECIALIST (SURGEON, OPHTHALMOLOGIST...) ENTER THE SPECIALITY  
-HOSPITAL STAFF DOCTOR (ARMY DOCTORS EXCLUDED) ENTER C-SA01  
-NON-HOSPITAL, STAFF DOCTOR (ARMY DOCTORS EXCLUDED):  
COMPANY DOCTOR, SCHOOL DOCTOR, CONSULTING DOCTOR ..ENTER C-SA02  
-INTERN ENTER C-SA23  
ATTENTION: DOCTORS OF DENTAL SURGERY AND  
ORTHODONTISTS ARE DENTISTS ENTER C-PL23

Verbal protocol: "One has to "enter the speciality"..."

I'll type "hospital staff doctor".

(Researcher: Didn't you enter the speciality ?)

I did ,but they didn' t accept it."

(B) Operator: profession = inspector  
coded economic activity = 9004 external administrative services other than economic or financial:

Computer: PIVOT WORD "INSPECTOR", SECTION SUPERVISOR (PUBLIC DUTIES)  
-INSPECTORS, POSTAL ENTER C-FP21  
-INSPECTORS, TREASURY, CUSTOMS, TAX ENTER C-FP22  
-INSPECTORS, STATE (VERY RARE) ENTER C-FP11  
-INSPECTORS, CIVIL AVIATION ENTER C-TD02  
-OTHERS ENTER C-FP24

Verbal protocol: "I think it ought to take the economic activity code into account because it has returned "inspectors, treasury, customs, tax," when that is a part of the entry "

(C) Operator: economic activity = food manufacturing

Computer:

6101 SUPERMARKET  
6102 GROCERY STORE  
6103 SUPERMARKET COMPLEX  
6211 INDEPENDENT RETAIL GROCERY STORE  
6221 FRANCHISED RETAIL GROCERY STORE  
6212 SMALL INDEPENDENT SUPERMARKET  
6222 SMALL FRANCHISED SUPERMARKET

COMMERCE, WHOLESALE: CLASS 57, CERTAIN AGENTS...(BROKER, COMMISSIONER, ) are in CLASS 60  
COMMERCE, RETAIL: CLASS 61 FOR THE LARGE STORES... AND CLASS 62 FOR SMALL AND SPECIALIZED STORES.

\*\*\*\*End of this designation.

Verbal protocol: "This is for...commerce ?!"

One would have expected something about production...  
strange!

If I enter "production" will it be the same ?!"

(D) 1. Operator: profession = doctor of urology

Computer:

DOCTOR OF MEDECINE..., DOCTOR...(WITHOUT A PRIVATE PRACTICE)  
-SPECIALIST (SURGEON, OPHTHALMOLOGIST...) ENTER THE SPECIALITY  
-HOSPITAL STAFF DOCTOR (ARMY DOCTORS EXCLUDED). ENTER C-SA01  
-NON-HOSPITAL, STAFF DOCTOR (ARMY DOCTORS EXCLUDED):  
COMPANY DOCTOR, SCHOOL DOCTOR, CONSULTING DOCTOR... ENTER C-SA02  
-INTERM ENTER C-SA23  
ATTENTION DOCTORS OF DENTAL SURGERY AND  
ORTHODONTISTS ARE DENTISTS ENTER C-PL23

Verbal protocol: "I suppose he's a specialist since he's a doctor of "urology".

And it tells me to enter the speciality (laughs). I 'll  
key in 'urologist'."

2. Operator: profession = urologist

Computer: SA01 HOSPITAL STAFF DOCTORS (HAVING NO PRIVATE PRACTICE : ARMY DOCTORS EXCLUDED)

Verbal protocol: "It's strange that it doesn't respond with "specialists"  
I don't understand."

(E) 1. Operator: profession = professor of textile design

economic activity = technical instruction

Computer:

POST SECONDARY, NON-UNIVERSITY INSTRUCTION  
JUNIOR COLLEGE, TECHNICAL COLLEGE, TRADE SCHOOL, AGRICULTURAL COLLEGE,...  
THE ECONOMIC ACTIVITY CODE IS INSUFFICIENT TO DETERMINE THE INSTITUTION IN QUESTION  
JUNIOR COLLEGE; EXCEPT INDUSTRIAL ARTS COLLEGES (IAC) ENTER C-EN09  
IAC AND OTHER TEACHING INSTITUTIONS ENTER C-EN07

Verbal protocol: "I am going to enter "and other teaching institutions."

The respondent didn't specify whether it is a junior college ."

2. Operator: economic activity = EN 07

Computer: UNKNOWN ECONOMIC ACTIVITY CODE

Verbal protocol: "Was I supposed to enter it under the firm's name ?

(Researcher: No ...the confusion comes from the fact that your message pertains to  
the profession.)

Should I enter it there, then? (he puts it in the profession zone) But it's  
not a profession, is it? I don't understand."



(F) 1. Operator: profession = inspector, business frauds  
coded economic activity = 9004 external administrative  
services other than economic or  
financial....

Computer:

PIVOT WORD "INSPECTOR", SECTION SUPERVISOR (PUBLIC DUTIES)  
-INSPECTORS, POSTAL ENTER C-FP21  
-INSPECTORS, TREASURY, CUSTOMS, TAX ENTER C-FP22  
-INSPECTORS, STATE (VERY RARE) ENTER C-FP11  
-INSPECTORS, CIVIL AVIATION ENTER C-TD02  
-OTHERS ENTER C-FP24

Verbal protocol: " Ahh! It's "others." "

2. Operator: profession = FP 24

Computer: FP 24 OTHER PERSONNEL UNDER CATEGORY B OF PUBLIC OFFICE (ADMINISTRATIVE  
SECRETARIES; INSPECTORS, WORK, TRADE,...)

Verbal protocol: "... (silence)

(Researcher: Does that surprise you ?)

No...but, after all! A moment ago, it said "others" and then ...Still, if that's  
the way it works! One has to read it in relationship with the message  
just received. Is that it? No, because read like that, it is "other personnel  
of category B." one asks oneself, "what's going on here!" Still, one must  
remember that...("Others?") Exactly"

For want of space I am unable to enter into a detailed analysis of these  
exchanges. It is easily seen, however, that the following conclusion may be drawn:  
if the operator is troubled by the responses of the computer, it is because he expects  
it to follow a principle of exchange that one might call, borrowing an idea from  
Grice (1975), the "cooperative principle." I would suggest some rules the operator  
assumes that the computer respects:

(a) Take account of all the information transmitted.

Violations: (case A) some information is not taken into  
account; (case B) information already transmitted is re-  
quested again.

(b) Be pertinent.<sup>-192-</sup>

Violation: (Case C) a response is given which has little bearing on what has been transmitted.

(c) Be logical.

Violation:(Case D) a failure to take the preceding step into consideration.

(d) Give all the information necessary in the exchange.

Violations: (Cases A and B) failures to indicate that all the information has not been used.

These rules are entirely similar to what can be described in human communication.

I can connect this fact to features of the interaction :

- The result of the operator's action (i.e., the computer's response to the transmission of a display form) is unknown for him. As a matter of fact, he does not know the contents of the files consulted by the computer . For the operator, therefore, it is not as if he were simply giving orders to a machine. He is faced with a kind of interlocutor. The computer's responses are utterances, either explicit ("The economic activity code is insufficient to determine the institution in question.") or elliptical (as when, for example, it produces lists of designations). Moreover, these utterances were initially composed by real interlocutors for the operator (i.e., those who developed the economic activity and profession classifications in order to transmit information to him and to compel a certain action.

- But if the interaction resembles a conversation in some respects, it is quite different in others: the computer's responses are fixed and sometimes are not well suited to the situation; they are produced in a mechanical way; the means available for interrogating the computer are poor (consisting only of an ability to modify the content of the zones). The operator, indeed, confronts a machine which responds to his orders. Even so, he expects the machine to be reliable, régular , systematic, The interaction then has an ineradicably double nature: it is at once both conversational and mechanical.

It is with this reservation that I would speak of "dialogue".

Notwithstanding the reservation, one must not overlook the fact that the operator supposes or seeks rules or, at the very least, regularities in order to define his conduct. When the rules are violated:

- either the dialogue malfunctions (in ways ranging from the "pathological dialogue" when the anomaly is serious (case E) to a simple incongruity ("irregular dialogue") when the breach of rules is slight);

- or the operator, taking it for granted that the response is consonant with the rules, will seek to discover something implicit in the response which gives meaning to it (case F).

On the basis of this analysis, I conclude that the interpretative activities of the operator are not only of a semantical character, but equally of a pragmatic nature. The computer's response is not a simple transmission of information, because it takes place in the context of what one can call, following Wittgenstein, a language game of a particular kind.

#### Work load and system design

Pathological or irregular dialogues lead to an increase in work load, as one sees by looking at certain indices that we are able to precise: parasitical supplementary activities, misleading induced reasoning, increasing burdening of the memory, ... An analysis of the sort that I've sketched enables one to make certain prescriptions. I will call that feature of the system which leads to these effects conversational incompetence. For the system that we have studied we can define precisely the elements of this incompetence and seek ways to reduce it.

These latter take several forms:

- (a) editing and presentation of the responses (Cases E and F)
- (b) modification of the files of pre-established designations (Cases A and B)
- (c) modification of the search algorithm for the file (Case B)
- (d) modification of the way in which the lists are constructed (Case C)
- (e) introduction of a marking operation for the display form of words and zones used by the computer in its search.

## 5. Conclusion

I have examined here only one aspect of the operator's activity , that concerning the dialogue with the computer. This dialogue is only a means for realizing the codification of information. As I've demonstrated in a previous study (Pinsky et. al., 1979), this codification is not a simple categorization; it is an authentic problem-solving process. The question arises as to what kind of connection exists between this problem-solving activity and the dialogue with the computer. The classical studies of problem-solving (from Piaget to Newell and Simon) have only considered what one might call the operating logic of the subject and not the interaction phenomena with the other participant of the dialogue. On the other hand, studies of human communication have been little concerned with the pursuit of a cognitive objective or the solution of problems introduced by constraints other than those of the dialogue.

As in every work situation, the one we have considered above is not "psychologically pure", and its analysis requires that one have recourse to several scientific disciplines.

In the field of computer design considerations of the problem solving activity suggests the need to build a second type of competence into computer systems: a competence which aids the operator in solving problem.

But this is another subject, which I must leave for another occasion.

## References

- Grice H.P. 1975 "Logic and Conversation", Syntax and Semantics, vol. III, Speech Acts, ed. P. Cole and J.L. Morgan , Academic Press,
- Pinsky L. Kandaroun R., Lantin G., (1979) Le travail de saisie-chiffrement sur terminal d'ordinateur, coll . Physiologie du travail et d'Ergonomie du C.N.A.H,n°65
- Smith H.T. (1980) "Human- computer Communication", Human Interaction with Computers ed. H.T. Smith and T.R.G., Green, Academic, Press
- Wittgenstein L. (1961) Investigations philosophiques, Gallimard, Paris

Inside and Outside the system,  
Problems of Communication, Interaction and Understanding in a  
Programmable Environment.

dr. A. Dirkzwager  
Vrije Universiteit Amsterdam

Summary

1. The inside of a system looks very different from its outside.  
Open up the hood of your motorcar and you see what I mean, or look behind the back-panel of your washing machine to discover the works of wires, switches and micro electronic components. Normally we are shielded from the complications of the inside of such systems and we can live with them and use them quite comfortably staying at the outside. This is not only true for artificial, technical systems but also for natural systems. We can have very interesting and meaningful discussions with a fellow-human being without bothering about firing neurons or the internal chemical processes that keep us alive and lively (if we would we probably would not be able to understand the meaning of the ongoing discussion any more). The same is true if we consider larger systems like an army at war where we can abstract from the internal individual soldiers with their guns, ammunition and technical equipment if we consider strategy and tactics, or systems like a large business organisation with its external policy which can be understood without detailed knowledge of its internal structure and functioning. The importance of the systems concept is that it draws a sharp distinction between the inside of a system and its outside such that we may either focus our attention on the internal functioning of the system and abstract from its outside meaning or focus our attention on the outside functioning and the outside meaning of the system without bothering about the internal specificities.
2. Communication is a process in which (at least) two systems interact in a common environment. It can be observed and understood from at least three different viewpoints, and at different levels. Two viewpoints take their position inside one of the communicating systems. In that case the other system is at its outside, in its environment. For instance man-machine communication may be designed and studied

from the perspective of the machine, man being part of its environment providing inputs and accepting outputs, or from the perspective of the man, under which perspective the machine is part of his environment reacting to his inputs and providing informative and to some extent useful output (in the limiting case: garbage). A third viewpoint is possible, that of an outside observer who considers the communication process as a whole, but as soon as he distinguishes two different communicating systems in a common environment he is bound to be biased by focussing his attention at one of the systems, and (depending upon the language chosen to describe this "fore-ground-system") by choosing a level of description for the interaction. In a man-machine system for instance he may describe the interaction either as an exchange of physical stimuli and responses or as an exchange and interaction of meaningful concepts and arguments. Much confusion is caused if one is unaware of differences in viewpoint and in focus of attention or if one tries to entertain two different viewpoints at the same time: it is like the perceptual background-foreground phenomenon: one can switch one's perspective, but it is impossible to see both sides of the boundary as the intended foreground figure at the same time.

The case for the "outside observer" is not very strong as it implies a distinction between this observer as the intended system, and his environment in which the two communicating systems are placed. The observers viewpoint is a privileged one and the problem is shifted to the question how communication is possible with another observer, who has his own privileged viewpoint that can not be switched without giving up his unique observing position. So the question which one of two interacting systems is regarded as the intended foreground system stays crucial.

3. Understanding is some kind of synchronisation between two communicating systems. When one system can reproduce the input-output relations of the other system it is said to understand the other system; a system shows to understand the other system when it responds and behaves in ways expected by this other system. In that way not only people can understand each other, but a man can understand a machine and we may even say that a machine "understands" its operator (or does not "understand" someone who does not know how to handle it properly, for that matter). The concept of "understanding" rests upon a strict distinction between two systems and a confusion of elements in one system with elements in the other system (which may be quite different) by calling them "the same" or by perceiving them as equal,

as if it were possible to do so with concepts under two different viewpoints.

"Understanding" can be described and realized at different levels and with different degrees of precision and detail depending upon the language used to describe the communication and the systems participating in the communication. A modern washing machine "understands" the language of the dials and switches that select a certain programme. It is a closed system that can be easily understood in the outside world at least as long as it is not broken. When it is broken its identity as a closed system is lost: it does not react any more as a proper washing machine and probably it produces funny noises from its inside that upset the outside world. In that state it "understands" only the language of pliers, screwdrivers and soldering irons and in its behavior (or non-behavior) it "speaks" a language that only the mechanic can understand. The machine has to be opened up and some communication in this lower level language is necessary to have it repaired and restore it as a closed system that can be understood in a high level language and that can be used without bothering about its internals. It is a matter of good design that machines are closed systems with a transparent interface to the outside world, transparent in the sense that its behavior can be easy and completely understood in a metaphorical way using common concepts from everyday language without knowing about its internal structure and functioning that can only be understood using scientific or technical language: in that respect it should be a closed black box. If it is opened up for repair or maintenance the same principle holds. What shows should be a set of black boxes with transparent interfaces that is intelligible to the maintenance engineer who speaks the language at this level. At still lower levels the internal physics of the components as a whole might be only understandable to the specialistic scientific engineer, who is not necessary the most knowledgeable in using the system.

4. To program is to use tools and machines (components) available in the environment to construct a new machine which serves certain purposes. A programmable environment offers the tools and components to do so. The process would be quite unmanageable and incomprehensible if not different system-levels are distinguished and kept clearly separated. (The development of) a modern computing system is a good example. It started with computing machines in which the programs were wired in

at plugboards. Using the tools and components of this (hardware) level, machines were build were programs could be input by setting switches or reading punched tapes or punch-cards containing binary code that was kept in core: the stored program computer. In this machine all the wiring (at least most of it) was inside the black box. Soon this machine was programmed (in binary code) in such a way that it became programmable in a mnemonic code, assembler, which was much easier to learn. The boundary of the system was moved such that the binary code moved inside the black box and the interface became more humane. Still one had to translate ones algorithms, formulated in flow cards or any other easily readable language, into a quite unreadable code. With the design of compilers for "high level programming languages" (in assembler-language) a machine with a wider boundary was constructed as the coding-part of the process was also put in the black box which became more humane and easier to program; a "general purpose" computer to develop new machines for ones own "special" purposes. With the uprise of operating systems and jobcontrol languages (often defined and constructed in higher level programming languages) the boundary of the system-as-used moved further such that less technical details on its inner working should be known and such that its outside boundary became more transparant to the user. In the mean time the scientists and technicians worked on the black boxes in the black boxes inside the system redesigning and replacing components to make the whole more efficient, faster, smaller, more powerful and cheaper. That is nice but of no concern here as far as it does not change the functional outside of the system, the "language" we have to use to understand and to use it: no one cares if the telephone-connection is realised by complexe mechanical relays or by electronic computers as long as the connection is realised reliably in reasonably short time (may be we should care, but that is another topic).

The view presented so far pictures the development as seen from the inside of the system. The intended system in the focus of our attention is at the machine-side of the boundary that forms the distinction between man and machine. As a consequence the perspective is that machines generate a demand for human work (wiring program-plugboards, coding in assembler language, punching tapes and cards) and that this human work is taken over in the next step by automation (it is absorbed by the next layer of the black box). The perspective of course leaves eventually little room for man's work as it is to be done automatically as soon as it is well-defined and it becomes



well defined as soon as its function in relation to the machine is clear. The crux is that from this viewpoint (the machine as the intended system) the environment is seen as machine-like and so is man in this environment if we focus our attention on the machine-part of the man-machine distinction.

Quite another perspective results if man is the intended system of our man-machines distinction. Then the machine becomes the secondary system of this distinction: it is seen as (part of) man's (programmable) environment. Under this perspective the emphasis is on the meaningful information processing by man, the distinctions he makes and imposes on his environment, the relations he discovers and constructs such that the behavior of things and machines in his environment become meaningful in the context of his own cognition and architectural activity in this environment. From this viewpoint not the machine but the man is the intended black box, the system that expands its boundaries. The emphasis is not on the syntax and the physical implementation of "information" processing, but on the semantics and the cognitive meaning, on man as a creative thinking and constructing creature who molds his environment according to his ideas and conceptions in such a way that these ideas and conceptions can be recognised in the environment by other subjects who are consciously living in it.

Originally the boundary of this system called man roughly coincides with his skin, it is extended as soon as man starts to use tools; the man and his tool is one system that operates upon the environment. By the invention of writing the boundary further moved: physical inscriptions are made and become meaningful not only as an extension of man's memory or as a facilitation of interhuman communication but also as a tool to enhance his information processing capability: extensive calculations become possible on paper using the right notational system; concepts, ideas and relations between them become understandable by making schematical drawings. Each expansion entails an extension of the original system called "man" with additional concepts, intellectual skills and technologies that change the contents of the original "black box" such that a more powerful "black Box" results. Subjectively it is often a difficult, sometimes painful process of adaptation to new technologies, of becoming literate and cultivate. This process however can also be understood - from the viewpoint of man as the intended system - as a process of assimilation and interiorisation of external functions, moving the boundary of the self such that these

external possibilities become part of it. One has to become familiar with the instrumental facilities to such an extent that one perceives the activity of the man-machine system one is part of as an activity of one-self: not the pen writes but the writer, not the car follows a certain road but the driver, not the agenda remembers appointments but its owner. This implies often that new conceptualisations have to be formed and that initially alien concepts have to become a natural part of the self. They imply a displacement of the boundaries between "self" (including one's own well understood environment) and "alien" (an external system, understood and not to be trusted, that largely determines one's duties and one's fate). Often this is quite difficult for grown ups who have a well-established selfconcept and knowledge about the world they are used to. Children much more naturally accept a new environment and grow into a natural feel of how to use and work with machines and artefacts available.

Young adults are often better able to survive in the competition than older ones, because they are better able to understand new developments as they are not hindered by old established concepts and distinctions. For this reason some of these changes may take at least one generation. Primary and permanent education should be of help but often the educators themselves are at a loss in the rapid development and they might be a hindrance by unconsciously imposing their attitudes and reactions which are based upon out-of-date conceptualisations. This implies from a systems-design point of view that one should carefully design the outside of the system in a way that it does not need "educators" for its use and that eventually even the educators can understand it. Programming languages and programming systems define such systems if they are well-designed.

A programmable environment is defined by some programming language which is a boundary or interface that separates man from this environment which he does not need to understand beyond this boundary, the programming language). With natural environments (including the physical world, other people and the artefacts they made) learning the language is quite a problem leading to the development of sciences and technologies. With artificial environments, for instance computer(based) systems, the interface should be designed in such a way that learning the language is no great problem. It should be such that it is an almost natural means to formulate things one wants this environment to do. Concepts and constructions used by man should be legal concepts and constructions in this programming language, at least as long as they are not contradictory

or too complex to be understood. Not all such legal concepts have to be understood by the environment immediately, but every legal concept or construction must be explainable in legal concepts and constructions that are understood by this environment. This "explanation" should be possible in the programming language in terms of concepts and constructions that are easily understandable to man acting in this programmable environment. On the other hand the programming language should provide the possibility of constructing a strict boundary between the original concepts and constructions and their explanation (examples are macro-calls and macro expansions, procedure calls and procedure definitions, methods of structural ("top-down") programming, progressive refinement, procedures of sending messages and receiving reactions from objects and the definition of these objects, labelling and naming of variables and their implementation with memory addresses).

The result of man's programming activity is twofold. On the one hand his original concepts and constructions (algorithms) are implemented in the programmable environment, the boundary between them and their explanation in the programming language define a new environment that can be understood in terms of the original concepts rather than in terms of their explanation or expansions in (machine) code, the (new) environment is more humane and transparent as the (lower level) code (even if formulated in higher-level-language programs) becomes invisible in the background. On the other hand this expansion of the environment with new information processing capabilities may be quite incomprehensible to anyone who does not, in his own language, understand the original concepts, constructions and intentions of the programmer. Digging into the code or the procedure definitions is hardly a solution as looking at the internals of a black box hardly makes its outside function and meaning any clearer. One should know beforehand what ideas are implemented. What is needed is a piece of the programmer's mind, the context that gives meaning to the concepts, constructions and ideas he implemented. It is the programmer's responsibility to make his programs humane, that is to say easily understandable to other people sharing the same cultural background (using suggestive names instead of incomprehensible code numbers is one method). It is the responsibility of education to provide this cultural background so that anyone coming into contact with these programmable/programmed environments has the necessary cognitions to understand them and make interhuman communication about them (a.o. with the programmers) possible. It is the responsibility of programming-

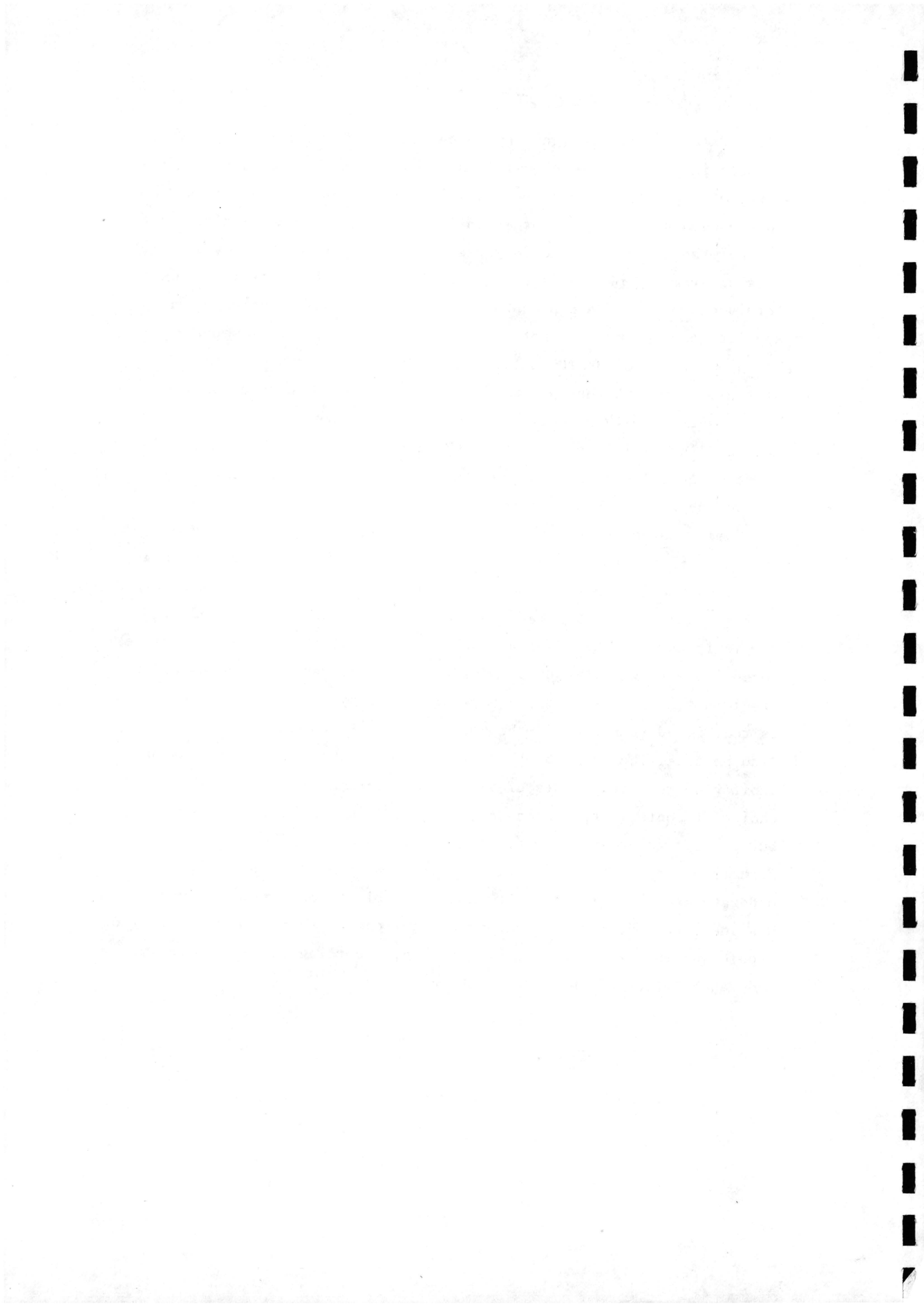
systems designers to develop languages that facilitate this interhuman communication. From this viewpoint the power and efficiency of the implementation of new programs is a matter of routine and good craftsmanship, the real problem is that the programming language (and the good programs formulated in them) should be understood by anyone involved in or influenced by the use of it. To design such languages and to educate people to use them properly in their communication with each other (and of course with their machines) is at the moment a most urgent problem, as systems should be understood.

5. Some consequences for the design of programs and programming languages shall be illustrated with examples from existing programs and languages. The program, defined in a certain language is seen as a boundary that separates man from the machine. At one side of this boundary we find man who has a more or less clear conceptualisation of the kind of machine he wants, he is able to formulate in his own words more or less precisely an algorithm for the behavior of this machine. On the other side of this boundary we find the machine (or the programmable environment) that offers the components and possibilities to combine them into larger structures from which the wanted machine can be constructed or, in other words, the programming language in which the wanted behavior can be defined in a way that enforces the machine to show this behavior. The elements of the programming language are most of the time easily understood by man, as is his concept of the wanted machine. The problem is that there is often a large distance between them that is not easily bridged by known relations and combinations. To bridge the gap is the activity called programming where bridgebuilding starts at both ends, "bottom up" by constructing bits and pieces of program that can be executed and are hoped to make the given machine look more like the wanted one, and "top-down" by explaining the original conceptualisation in terms of concepts and subalgorithms that one hopes are easier to program using the given bits and pieces produced in the bottom-up way. Programming languages favor the bottom-up approach most of the time, even well-structured languages like Pascal where variables and procedures have to be declared and defined before they are used. In the top-down approach their use in a larger context comes first and their definition ("coding") in terms of the programming language is postponed and can be postponed when the programmer can clearly and understandably define in his own words what the variable stands for or what effects a procedure has if it were implemented.

Reference -203-?

In ELAN (a language designed by Koster and used at many secondary schools in Germany) one can write a legal program using names for parts of the program that are not yet defined, the definition is postponed and seen as a (not essential although necessary) elaboration of the meaning of the name in terms of the programming language. This programming method is called "refinement". Of course it is not forbidden to define a procedure beforehand when one foresees useful applications, but one is not forced to do so. In the same sense it is an advantage of BASIC that one can introduce (names of) variable at the very moment they are needed, type-checking can be automatised as the variable type becomes clear sooner or later out of the context and conflicts still can lead to error-messages. Sometimes the type is not even relevant at a higher language level, for instance when two variables of the same type have to exchange values.

Some remarks on research regarding the ergonomics of programming languages remain to be made. This research is very important in order to discover with (constructions available in a language), the limitations of the human mind to handle complexity and to discover the kind of complexities that should be avoided. It is for instance quite possible to design a transparent program using goto-statements for certain problems that lead to complex programs not easy to unravel if one is restricted to nested if-then-else constructions. The reverse can also be true. The sense of this kind of research is not to act as an arbiter between proponents of differing programming styles and the choice of control-structures to be admitted in a programming language, but to give guidelines for a disciplined use of the different control structures according to the kind of problem at hand such that understandable programs result, and to inspire programming language designers to implement a programming environment that enhances efficient problemsolving procedures in programming, according to the limitation and capabilities of the human problem solver.



## Naming Commands: An Analysis of Designers' Naming Behaviour

Anker Helms Jørgensen	Phil Barnard & Nick Hammond	Ian Clark
Institute of Datalogy Copenhagen University Sigurdsgade 41 DK-2200 Cop. N Denmark	MRC Applied Psychology Unit 15 Chaucer Road Cambridge CB2 2EF U.K.	Human Factors Lab. IBM United Kingdom Labs. Hursley Park Winchester SO21 2JN U.K.

### Abstract

Users often experience trouble in learning and using the commands in interactive computer systems. They claim, for example, that the command names are computer-centric and that the abbreviations are unsystematic and unintelligible.

In order to substantiate the grounds for such claims a study of computer system designers' naming behaviour was carried out. An interactive questionnaire was shipped on a computer manufacturer's communication network. The questionnaire simulated a simple message-decoding system. The decoding took place in two stages. First the details of the message-transmission procedure were established in 4 function steps (e.g. establish the transmission wavelength). Next the message itself was decoded in 8 function steps.

The task of the system designers was to name the functions carried out in each of the 12 steps. The functions were presented as "before-after" display frames. The questionnaire was filled in by 110 designers.

The most striking feature of the result is an incredible variation in the designers' names: every other designer chose a unique name. However, an archetype appears across all the names: 68% were non-abbreviated single-word names.

When the names for each function are analysed semantically across the 110 designers a clear consensus emerges despite the large superficial variation. Object-oriented names were chosen overwhelmingly for functions generating new information, addressing the new information. The designers' selections were clearly influenced in these cases by the task information available, e.g. table headings. In contrast, action-oriented names were chosen overwhelmingly for functions generating a new version of previously displayed information. Computer-centric names were quite frequent in these cases.

Hence a semantically archetypical command set appears. About 1/3 of the designers adhered to the archetypical command set. However, very little consistency appeared when a single designer's choice across the 12 functions was considered. Inconsistencies in for example abbreviation schemes and name structure were the rule rather than the exception, even among the designers adhering to the semantically archetypical command set.

In conclusion, the claims mentioned initially are to some extent supported by this study. More specific, two points emerge. First, a large degree of consensus semantically between the designers is blurred by an incredible superficial variation. Second, most of the designers show a fairly small degree of systematicity within the command set.

Paper to be presented at COGNITIVE ENGINEERING  
A Conference on the Psychology of  
Problem Solving with Computers  
Amsterdam, 10-13 August 1982







