

This paper should be cited as: Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge, UK: Cambridge University Press, pp 443-460.

Cognitive Dimensions of Notations

T. R.G. Green

MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge CB2 2EF, UK.

‘Cognitive dimensions’ are features of computer languages considered purely as information structures or notations. They therefore apply to many types of language—interactive or programming, high or low level, procedural or declarative, special purpose or general purpose. They are ‘cognitive’ dimensions because they control how (or whether) the preferred cognitive strategy for design-like tasks can be adopted: it has repeatedly been shown that users prefer opportunistic planning rather than any fixed strategy such as top-down development. The dimension analysis makes it easier to compare dissimilar interfaces or languages, and also helps to identify the relationship between support tools and programming languages: the support tools make it possible to use opportunistic planning with notations that would otherwise inhibit it.

Keywords: Computer Languages; Opportunistic Planning, Cognitive Dimensions.

Introduction

“Not another study on usability”: there are too many reports, not enough powerful generalisations. This paper offers a generalisation which is not yet fully worked out but promises, if successful, to be powerful.

Physicists reduce all physical quantities to combinations of three fundamental dimensions, mass, length, and time, all of which are abstractions from the phenomenal world (for example, we perceive weight but not mass). Such varied phenomena as electric charge, gravitational attraction, elasticity, and temperature are all reduced to different combinations of the same three dimensions: a remarkable and enviable achievement. Rather cheekily, I suggest that in HCI we attempt to tread the same path. Many different ways of working at the computer can potentially be accounted for by the interrelationships between a single preferred cognitive strategy and a small number of facts about the language of communication, or ‘notation’, and the circumstances of its use, or ‘environment’. These few unifying facts are the equivalent of physicists’ dimensions: only a very approximate equivalent, because where cognition is involved, we are unlikely to find simple orthogonal concepts such as those that physicists have, over the course of many years, refined so elegantly. Cognitive science, it must be remembered, is really rather more like plant biology than like physics, in that a whole host of factors (some of them interdependent) affects the growth of plants. Nevertheless our urgent need is for a small number of clear, powerful ideas, which we can pretend are orthogonal.

To arrive at the suggested dimensions, I shall consider a ‘space’ in which to locate many different kinds of notations, including command languages, direct manipulation languages, programming languages, and other types of notation, and I shall describe some ‘cognitive dimensions’ of this space, illustrated by concrete examples. The dimensions provide a language in which to compare the *form* and *structure* (rather than the content) of notations.

We are more used to speaking of programming ‘languages’ than notations. I shall use the term notation to keep the form distinct from the content. Pascal, as a programming language, might be properly criticised for its content (poor provision for string manipulation, bit processing, and file

handling); but as a *notation* these omissions are not relevant—what matters is the form; e.g., the identifier hierarchy is very rigidly controlled, from which both advantages and disadvantages flow.

The following sections introduce the ideas of information structures and their ‘dimensions’ and say a little about their interaction with cognitive processes. Some sample dimensions are then discussed; this is the core of the paper, although space prevents going into detail, or even discussing more than about half the relevant dimensions. Finally, to prove that the concept of notational dimensions has design relevance and is not just a vacuous abstraction, these ideas are briefly related to the fashionable problem of designing usable OOPS systems.

2. System = Notation + Environment

The importance of the relationship between the notation and the support environment must be emphasized. Here is an example. A few years ago, a team developed a system for receiving spoken Pascal code, using an isolated-utterance speech recognizer. Intended for use by handicapped people who had difficulty in writing or typing, the system worked reasonably well—i.e., if one dictated Pascal code to it, the recognition rate was quite acceptable. However, because the speech recognizer relied upon the constraints of Pascal to make the recognition problem tractable, the program had to be syntactically correct at all times. The design of the system made it preferable to dictate the program in text order, from start to finish. Unfortunately, because of certain characteristics of the Pascal notation, it is pretty well impossible to dictate impromptu Pascal correctly without any omissions. (See below.) In short, although the system worked reasonably well at the technical level, it failed to meet its objectives, simply because *Pascal is not a suitable notation for this environment*. A suitable environment for Pascal needs, as we shall see, to decouple the order of generating source code text from the final text order. If the relevant characteristics of Pascal and the speech-driven environment had been appreciated beforehand, the project might have taken a different course.

Indeed, the relationship between the notation and the environment is such that the notation cannot be used except in some kind of some kind of environment of use. Trying to just use a notation, outside of any environment, would be like trying to just talk to someone without being in some kind of social situation. We may catch ourselves thinking that paper-and-pencil, by its familiarity and blandness, is not really an environment at all, but if so we are making a mistake: paper and pencil is an environment with its own particular contributions — for instance, it offers unrivalled support for recording hesitations and commitments, makes it easy to see large amounts of text with little effort, and supports instant action with very little delay in finding the right place, choosing the right mode, etc. In all these respects it is superior to typical computer-based environments.

Which part of the system, then, is the notation, and which the environment? If we change the system, which have we changed? This question is too large to analyse here in depth. As a rule, certain parts of the system are obviously notational, while others are obviously environmental.

Ambiguities will occasionally occur, especially when the symbols displayed as code or text do not correspond fairly directly with some subset of the user’s actions. If we have function keys to generate syntactic constructions, for example, which is the ‘notation’—the keys we press, or the words we see? Various factors will determine the user’s view, such as prior experience; the units operated upon by the editor and other manipulation systems; and whether a simple mapping can be perceived between the two sides, so that a function key can be seen as generating a simple indivisible unit of a few words. But I hardly think these need deter us.

The fundamental principle is that the way the user behaves is determined by both the notation and the environment. A satisfactory system demands an environment that supports the notation and vice versa. Hence the slogan heading this section.

3. Designing Notations for Design

The ideas presented here apply to written-down, symbol-based systems such as conventional programming languages, school algebra, powerful document formatting systems such as T_EX and

Scribe, and also to the many types of fringe programming language: languages for numerically-controlled machine tools; programmable video editing systems; languages for knowledge engineering; executable specification languages, etc. Visual programming languages are particularly interesting notations.

They also apply to interactive languages. Sometimes these have the same structure, and differ only in the environment of use: “ $(4 + 6)/2 = 1$ ” is school arithmetic when written down, but it is also the interaction language for a calculator. The notational structures of interaction languages can be described in the same terms as those of programming languages, but because their instructions or commands are performed immediately the cognitive requirements are different. In the present paper, however, I shall give primacy to written-down notations.

Notations have manifold uses. Important ones are *communication* (at a distance or over time) and *design*. I shall focus on their use in design, for recording partial decisions, working out consequences, undoing or confirming decisions, leaving reminders, etc. In this context their role is partly retrospective, recording what has been thought out, and partly a means of discovery: “How can I know what I think until I hear myself say it”?

So the problem before us is *how to design notations for design*.

4. Notations as Information Structures

Notations have a structural aspect and a typographic aspect. Though both are essential in conveying meaning to the reader, I shall only consider the structural aspect, with its mainly discrete attributes, rather than such typographic issues as relative size and spacing of symbols, subtle and difficult questions dealing mainly with continuous quantities. (Some idea of the enormity of my omissions is given by Southall (1988)).

Very different notations can achieve identical ends: the same algorithm in different programming languages, the same graphic design from different drawing programs. Where the information structure is different, different cognitive processes will be facilitated. Take the case of reading programs. We cannot simply claim that procedural languages are either easier or harder to read than declarative ones; it would be more valid to say that the structure of procedural languages makes it easier to extract sequence information than circumstance information (Green, 1977), while the reverse is true for declarative languages (Gilmore & Green, 1984).

Different diagrammatic notations also favour different tasks (Green, 1982). For example, Nassi-Shneiderman diagrams and Jackson diagrams present the same information, but their different structures mean that it is much easier to make small changes to Jackson diagrams. Or take the interactive case: a familiar example is the contrast between reverse polish calculators and infix calculators, each good for some tasks.

Each notation highlights some types of information at the expense of obscuring other types; each notation facilitates some operations at the expense of making others harder. *A notation is never absolutely good, therefore, but good only in relation to certain tasks.*

5. Opportunism: Design is Redesign

The naive theory of action is that the actor translates a set of goals into specifications for the corresponding action sequence, and then runs off the actions. As applied to writing a program, a technical paper, or a graphic design, the implication is that actions proceed smoothly from start to finish without errors or omissions. (Any reader who seriously believes that should try dictating Pascal.) In actual fact, the creation of any such complex object proceeds in fits and false starts, with repeated iterations, and any goal or subgoal may be attacked at any moment, whenever suitable circumstances happen to arise. This has been shown in technical writing (Flower and Hayes, 1980), in CAD (Whitefield 1985), in system design (Carroll & Rosson, 1985; Carroll, Thomas & Malhotra, 1979; Guindon, Krasner & Curtis, 1988), in program design (Siddiqi 1985; Visser 1988), in the coding activity of novices (Gray & Anderson 1987), and even at a level where you might not expect it, *viz.* professional programmers coding very simple programs (Green, Bellamy & Parker, 1987). Hartson and Hix (1989) refer to ‘alternating waves’ of bottom-up and top-down activities in interface design work. Streitz (1988) has observed that ‘writing is

rewriting'; more generally, one could claim that design is redesign, programming is reprogramming, etc.

The single preferred cognitive strategy in design (judging by present research) is *opportunistic planning*. High level and low level decisions are mingled, commitment to possibilities can be strong or weak, development in one area is postponed because potential interactions are foreseen, parts are frequently re-evaluated and modified.

There are programming tasks for which this dictum may not hold, such as the transliteration of precise functional specifications into equivalent code. But in general, any growing structure, whether a program, a technical paper, or a piece of graphically-represented CAD, will need to be modified and rebuilt. In the old cybernetic terms, there has to be a feedback loop for error correction. In contemporary terms, we must not overlook the 'gulf of evaluation' (Norman 1986) and the cyclical nature of user activity in Norman's model.

If the notation and the environment together are to support opportunistic planning, three strong corollaries follow. First, since programmers are to be able to attack any goal at any level, they must be able to pick out what parts of the program support each goal or subgoal. Moreover they must also be able to make comparisons between any arbitrary parts of the program. To support opportunistic planning, therefore, it must be possible to *perceive the structure* of the code, text, or other notation. Second, if design is redesign, the notation must be *modifiable*: any attempted solution to any goal may be subject to later change. There is no room for an environment that makes it hard to go back and improve on a first attempt, nor one that insists that one part cannot be started until another part is finished. The twin activities of parsing notations (reading for structure) and modifying them are inescapable. The third corollary is in a poorly-explored area: since early stages of the design represent a wide band of possible developments, the notation should differentiate between degrees of definiteness. (Typographers use light pencil marks to indicate a wide band of possibilities—a font something like this—and firm lines for more definite choices. As Hewson (1987) observes, you can't do that with computer design tools.)

The most obvious failing of traditional notations, considered in this light, is that too little attention is paid to readability. It is sometimes brusquely asserted that "real programmers" can get used to anything, given a little "syntactic sugar", and that worrying about readability is namby-pamby stuff. It is indeed true that some programmers can cope with very demanding notations, and it is also true that some chess grandmasters can play twenty games blindfold simultaneously. A conscious design decision to cater only for such prodigies could not be faulted, but it would be stupid for a notation designer to ignore readability in a notation designed for widespread acceptance.

6. The Organization of How-to-do-it Knowledge

To understand the usability of notations we must understand how they are mentally perceived and organized. For interactive languages, the conventional wisdom refers to task/action mappings (Young, 1983) or to goals and methods (Card, Moran and Newell, 1983); for programming languages, the equivalent is the goal/plan analysis (Soloway and Ehrlich 1984). On this view, the writer or programmer decomposes a task into subtasks, etc, and translates subtasks into sequences of actions. Thus the notation is perceived and mentally represented not as individual units or symbols but as organized groups or chunks.

Ideally, we would know much more about mental representations than we do at present. Expert users know much more about their notations than task/action mappings: which bits are risky, time-consuming, etc. These pragmatic aspects strongly influence expert programmers' choices of languages (Petre, 1988). Also, in view of the importance of mental representations, it would be nice to know much more about the mental representation of a wider variety of interactive languages and programming languages. The work of Soloway and his colleagues, Détienne (1989) and others has told us much about the mental representation of Pascal, but how are very different languages, such as Prolog or Miranda, represented?

(A further limitation is that analyses such as GOMS give too little attention to 'situated action' and the possibilities of 'partial planning' (Young and Simon, 1987), or to the interpretation of the system image.)

However imperfect the conventional wisdom may be, it clearly implies that must be possible to comprehend the partial design *in the appropriate mental chunks*.

7. Some Dimensions

Broadly speaking, we need to be able to write notations, read them, and change what's written. In the present attempt to pick out determining factors, a 'cognitive dimension' of a notation is a characteristic of the way that information is structured and represented, one that is shared by many notations of different types and, by its interaction with the human cognitive architecture, has a strong influence on how people use the notation and affects whether the strategy of opportunistic planning can be pursued. It would hardly be possible to do justice here to an exhaustive list, but I shall briefly review some which are particularly pertinent to present interest in OOPS.

7.1. Hidden/Explicit dependencies

To illustrate the idea of a dimension I shall take three very different-seeming notations, and show that their structures share one very important characteristic which has great influence on how they are used. Subsequent dimensions will be treated more cursorily.

7.1.1. Spreadsheets

Figure 1, a and b, illustrates the structure of a typical spreadsheet. The notation explicitly shows the local dependency of cells, but it does not show either the long-range dependency (C depends on B depends on A ...) nor the inverse relation (A is depended on by B).

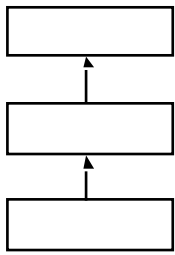
 <p>(a)</p>	<p>A</p> <p>22.45</p> <p>B</p> <p>= sin(A/2)</p> <p>C</p> <p>= B*B</p> <p>(b)</p>	<p>A: gene va 10 + le ft flush</p> <p>B: A + bold</p> <p>C: B + ce ntered</p> <p>(c)</p>	<p>A = (re d, blue, green);</p> <p>B = array [1 .. 4] of A;</p> <p>C = array [1 .. 2] of B</p> <p>(d)</p>
--	---	--	---

Figure 1 (a) A typical information structure, found in (b) spreadsheets, (c) style sheets, and (d) type declarations. In each case, the notation shows clearly that B depends on A, and C on B, but neither the long-range dependency of C on A or the inverse relation that A is depended on by B are shown.

Is this design decision important? It depends on what tasks need to be performed. If the user is errorlessly translating an algebraic model into spreadsheet terms, neither the long-range dependency nor the inverse dependency will need to be accessed. But one can very easily envisage the user of a large and ill-understood spreadsheet making a small modification which generated unforeseen, and possibly undetected, side-effects, because the cell that was changed was depended upon by other cells that the user did not observe. The only way to find out for sure what cells depend upon a given cell is to exhaustively examine every formula. Although there are few studies of how people use spreadsheets, it is notable that Brown and Gould (1987) found that experienced users spent no less than 42% of their time just moving the cursor around, mainly to inspect other cells.

Of course, the spreadsheet environment could easily complement this aspect of the notation by including a cross-referencer. Why is a cross-referencer not supplied as standard? Presumably because the designers did not believe that design is redesign. (Brown and Gould briefly mention an experimental interface in which cells referenced by a formula could be highlighted, and they claim, unfortunately with no details, that it improved performance; but there were other changes to the standard design, so we cannot be certain that their highlighter was responsible.)

7.1.2. Style Sheets

Microsoft Word supports ‘style sheets’ which determine the layout and font of sections of a document. All text that is ‘heading’ style can be formatted in the same way, and if the definition of heading style is changed, all the headings will be uniformly changed. For certain purposes this is extremely convenient. The styles form an inheritance hierarchy in which each style starts all characteristics of its parent, except those that are explicitly changed. The style sheet window of Microsoft Word 3, Macintosh version, gives exactly the same information as a spreadsheet, which is the local dependency (Fig. 1c). We can see that style B depends on (inherits from) style A, but we cannot see the long-range dependency (what does style A depend on?) nor the inverse relation (what depends on style B?). Thus if style A is changed, we cannot predict the consequences without gathering more information — and once again what is needed is an exhaustive search of the styles to discover which ones depend on A;

7.1.3. Type Declarations

Pascal, like many other programming languages, supports declarations of data types and structures in terms of other types and structures. Figure 1d shows how Pascal, yet again, exhibits only local dependencies in the information structure; long range dependencies and inverse dependencies are hidden. While this is no big deal in a trivial fragment of code, the problem is said to be acute in large programs; and various hierarchical browsers have been designed specifically to overcome it—illustrating nicely how the support environment can complement the notation.

7.1.4. Hidden Dependencies as an Abstraction

These three examples share a common feature: recovering the mental representations from the notation is difficult, because the information structure of the notation is both limited and asymmetric. This fact will interfere with opportunistic planning in each and every case, since it violates the requirement that the structure must be easily perceived. A slightly fuller analysis shows that the inverse ‘is depended on’ relationships of Figure 1 are not quite as bad as the long-range dependency case. In dissecting cell or line C of Figures 1b to 1d we know that there is one, and only one, thread to be found, and at each step we know the target. But answering the question about cell or line A, “I wonder whether this value is used anywhere”, requires a comprehensive search in which the target is only specified in one respect. Cognitive processes that are data-driven can use the former case as a reminder, but not the latter.

7.1.5 Implications

The lesson from the examples above is clear: every dependency that matters to the user should be accessible in both directions. Environmental support, in the form of cross-referencers, browsers, etc., should be supplied as a matter of course.

7.2. Viscosity/Fluidity

A viscous medium, in the language of hydrodynamics, resists local changes. (You can’t move a spoon quickly in treacle, because the treacle resists changing its shape.) Correspondingly, a viscous notation resists local changes. The problem occurs when the information structure contains many dependencies between its parts, so that a small change requires many consequent adjustments. One example is a text containing section numbers, figure numbers, and cross-references; either inserting or deleting a section may mean that many such numbers must be changed. Solutions are to avoid numbered sections (i.e., change the notation) or to use a knowledgeable document formatter, such as Scribe or T_EX (i.e., provide environmental support).

Whitefield (1985) reports similar observations in CAD situations. The degree of viscosity of a notation typically differs for different manipulations. Inserting a new formula cell into a spreadsheet is very simple, but rearranging the layout is quite another matter. Adding a new procedure to a Pascal program is usually straightforward, but reconstructing the identifier hierarchy so that a given procedure is brought to a lower block level, to make it more widely accessible within the program, may have many repercussions.

7.2.1. Implications

Viscous systems cause more work for the user. They also break the line of thought and create a working memory load if the user is to keep track. Yet they often have advantages. Their relatively high redundancy helps to detect certain errors—for instance, in programming languages where identifiers are declared, mistypings cause fewer problems—and sweeping accidental changes are virtually impossible. In some cases a notation is deliberately and knowingly made viscous by features which may cause extra work for the writer, but which are intended to promote readability. Indeed, the extra work involved may encourage users to think out their requirements in advance rather than doing ‘evolutionary’ programming or casually hacking.

When the notation is necessarily viscous, one solution is the knowledgeable environment, such as Scribe, already mentioned. Another solution is an ‘agenda’ system, in which the potential implications of the user’s actions are noted and kept available by the system. There are also ways to manipulate the notation, rather than the environment: a frequent treatment for viscosity is to augment the notation by introducing modules. Viscosity is high within the module but is low between modules, so the modules act to parcel up the viscosity into locally-sticky patches.

7.3. Premature Commitment

Imagine a word processor which, when first invoked, required the user to state exactly how many pages of text were to be written. That would be premature commitment. (Some early desktop publishing programs were not far short of this apparently absurd supposition: you had to lay the page out first, then pour the text in.) It can easily happen in programming, where the environment encourages the programmer to develop programs in the final text order rather than in the mental order of generation—the order in which the instructions come to mind.

What determines that generative ordering? In one of the most detailed models of cognitive processes in programming, Rist (1986) postulates a goal and subgoal generation process that depends on the programmer’s experiences and the familiarity of the solution components. “Novices are characterized by a forward development of the program with little planning ... Experts can define a particular part of the processing as most important, the focal segment, and accrete a program around this segment” (p. 31). Experts, according to Rist, “have both plan templates and sophisticated plan building mechanisms”. Rist’s example is finding an average. The expert might build the program by starting out with the ‘focal line’, which achieves the task of outputting the average; then work back like this:

1. output average; *this requires ...*
2. calculate average; *this requires ...*
3. count numbers and sum numbers; *these require ...*
4. read numbers; *this requires ...*
5. a loop.

This sequence bears little relationship to the sequence in which the instructions eventually have to be written in a typical procedural programming language. In general, the whole thrust of opportunistic planning is that *actions may be thought of in any order*.

No problems can arise in an environment where the statements may be inserted in any convenient order—paper and pencil, for instance. But where the environment restricts the acceptable orderings, the problems may be severe. Consider, as an extreme example, the spoken-Pascal machine described above. The programmer has to convert the mental generative order into an acceptable instruction order. Doing it by lookahead (i.e., working it all out in the mind before starting to speak) will impose a very high load on working memory. The only alternative is to risk premature decisions. Hoc (1988) demonstrated the difficulties faced by programmers in such situations.

7.3.1. Implications

What is needed is to *decouple the generative order from the final text order*, allowing any part of the design to be developed at any time. The need for decoupling has never been explicitly recognised in the development of programming tools: on the contrary, structure-based editors

frequently impose severe restrictions on the order of developing a program. The allowed orderings correspond to ‘top-down’ development, as a rule, and the authors sometimes claim that this reflects the ‘natural’ or best way to attack a problem (e.g. Kohne & Weber 1987) although Rist’s results for experts are pretty well the reverse of top-down ordering at the level of local, or tactical, decision-making during coding. (The claims for top-down development may, nevertheless, be correct at the strategic or global level.)

7.4. Role-expressiveness

It was shown above that the author/programmer must be able to recreate the appropriate mental chunks from a scrutiny of the code. Different notations facilitate this ‘parsing’ to different degrees. Notice that we are not parsing for syntactic structure but for *mental* structure—the reader is attempting to discover the role of each component, by finding a ‘plan’ or ‘schema’ to fit it into. Notations which display their plan structure clearly I shall call ‘role-expressive’.

(In a fuller treatment we would have to observe that different components of a structure can serve different roles, and may indeed serve several roles at once; also that there are many kinds of mental structuring imposed on a typical program, all at the same time.)

Brooks (1983) suggested that readers of programs looked for ‘beacons’ which indicate the presence of particular program components. Pascal’s rich set of keywords may well serve as beacons to help parse its structure, giving it relatively high role-expressiveness; even so, Gilmore and Green (1988) showed that if each hypothesized mental chunk was differently coloured, plan-related types of bugs were more readily spotted, indicating that the role-expressiveness of conventional Pascal could still be improved.

Since the reader of a document has to recognise the intentions from the code, the presence of keywords or other beacons reliably associated with particular intentions is almost certain to be helpful. An example of a beacon is an instruction sequence which only occurs in one context, such as a sequence to exchange the values of A and B — a pretty clear signal that some type of sorting is going on. Beacons are ineffective if the mapping from goal to instructions is many-to-one — if many different types of goal translate into very similar types of instructions; studying the instructions does not readily reveal the goals which led to them. Role-expressiveness demands that each goal translate into an action sequence containing some relatively unique indicator, a situation known in the psychology of learning as ‘cue validity’.

The common flowchart is a very familiar example of a notation with low role-expressiveness. Each configuration of actions and decisions must be studied to decide whether it is a conditional or a loop structure; higher-level judgements take still longer. Pure Lisp has the same problem with respect to data structures, since they are all—whether lists, trees, arrays, or records—built from the same list operations, merely assembled differently. Prolog programs set an interesting problem to the reader, because there is no indication whether data is flowing into or out of the rule.

The worst problems probably come when the material of each mental chunk is dispersed. Figure 2 illustrates data from Green, Bellamy and Parker (1987), showing what we believe to be two Prolog chunks, a filter chunk and an increment chunk. (Prolog specialists often dispute the nature, possibly even the existence, of Prolog chunks, but given that one putative chunk was written, and then after a pause the other putative chunk was added, it seems hard to believe that these are not the mental units in use by that programmer—who, it should be added, was a professional Prolog programmer.) It can be seen from the figure that the material of the second chunk is dispersed through the first chunk in a way that gives little indication of its nature. To recognise it as an increment, the whole chunk has to be apprehended — the same problem as the flowchart but on a bigger scale.


```

analyze ( [ ], 0, 0, 0, 0 )

analyze ( [1|T], 0, P, L, N ) if analyze (T, Pa, P, L0, N0 )
and L=L0 and N=N0 + 1

analyze ( [2|T], Pa, P, L, N ) if analyze (T, Pb, P0, L0, N ) and
Pa=Pa + 1 and longest (Pa, Pb, P) and L=L0 + 1

```

Figure 2. A fragment of a Prolog program written by a professional programmer (from Green, Bellamy and Parker, 1987). The parts in boxes (which together form an ‘increment chunk’) were written last and were inserted into the earlier code, suggesting that the subject’s mental representation comprised two plans, ‘filter’ and ‘increment’. Because the fragments of the second plan are dispersed throughout the first plan, and because there are no lexical keywords to use, it is hard to recover the mental plans purely from the text.

7.4.1. Implications

Role-expressiveness may be one of the most delicate notational decisions. It is seemingly bought at the cost of other virtues, especially uniformity, learnability and re-usability. A rich set of keywords or other beacons should make the code easier to parse back into goals, but will obviously tend to make the language non-uniform and will increase the vocabulary to be learnt.

Modularisation, introduced above as a treatment for viscosity, can also affect role-expressiveness. Typically, every module is designed to serve one structural role. Supporters of object-oriented programming claim that the modules imposed by an OOPS form a structure which has more cognitive meaning than the structures created by top-down techniques.

A possibility to be explored is the use of automatic highlighting of segments that appear to form ‘chunks’ or ‘plans’. Some investigations have already been made using ‘slices’ (Weiser & Lyle 1986), but the automatic slicing tool proved disappointing in practice. However, much depends, presumably, on correctly determining the segments to be highlighted.

At a higher level, Soloway et al. (1988) suggest a documentation style designed to give explicit indication of plan structures even when ‘delocalised’ and fragmented.

7.5. Hard Mental Operations

All the above has been written with a hidden assumption that all mental operations are equivalent in cost. This is an honourable simplification, hallowed by custom, but it is certainly not true. Certain notations require users to perform operations that are notoriously difficult. Familiar examples include the EVAL/QUOTE puzzles met in Lisp; the problems of pointers and indirection met in C; ‘off-by-one’ boundary condition errors, and the problems of multiple negation; of these, only the last has received much attention from cognitive psychologists. Sime, Green & Guest (1977) argued that one source of difficulties in comprehending certain types of conditional structure was the need to accumulate implied negations.

The cognitive processes underlying these familiar problems have not been analysed. Until then, it will be difficult to make useful generalisations, apart from enjoining notational designers to beware of the problem.

7.5.1. Implications

Perhaps tools can be built for some of these difficulties. Green and Comah (1984) attempted to build a ‘programmers’ torch’ which would answer circumstance questions about Basic programs: “Under what conditions can line 370 be executed?” Other researchers have designed tools for constructing invariants of programs, no doubt with the same idea in mind. Certain C statements are also puzzling, especially those using pointers, and one can imagine a C support tool that would give a diagrammatic elucidation of a statement using pointers. But no systematic attack is feasible until some basic research into the source of the difficulties has been made available.

7.6. Other Dimensions

Other dimensions not mentioned include *diffuseness* (APL v. Cobol); *consistency* (Payne and Green 1986); susceptibility to low-level errors, either in reading (the *discriminability* problem) or in action (the *action slips* problem); and the presence of *perceptual cues* to structure.

8. Cashing the Ideas

Abstract classificatory schemes need to be justified in concrete terms. The dimensions concept can be justified by pointing to areas where progress is slow, as I have done above; or by criticising existing systems; or by raising design issues in relation to present growth areas. Let’s take the last, and look at OOPS. Object-oriented systems have been praised as resolving some cognitive problems (Rosson & Alpert 1988), at least in principle. I take no issue with that, but are they adequately usable yet?

A fundamental claim in this paper is that *system = notation + environment*. Research papers on OOPS, however, offer a myriad new languages, with different ideas about the semantics of inheritance, etc., but say very little about environments. Here are some questions arising from the preceding discussion, with remarks relating to one OOPS, Smalltalk-80. I shall take the five dimensions I have described in order.

1. *Are there hidden dependencies and one-way links in the structure?* Smalltalk-80 scores fairly well here; most relationships can be browsed in both directions, although there are times when it would nice to make dependencies more immediately visible without having to search for them, so that they could act as reminders. However, there is little support in searching for ill-specified targets. “I want something that handles a bitmap, what can I find?” Nor is it easy to find out what kind of object can fill an instance variable slot in a method.
2. *If the inheritance hierarchy viscous, or is it easy to reconstruct in a different fashion?* The inheritance structure tends to be viscous. For instance, given a class `Animal` with many sub-classes `Trout`, `Herring`, `Minnow`, etc.. inserting a new class `Fish` between `Animal` and its subclasses requires many operations (add a new subclass, `Fish`, to `Animal`; move `Trout` to `Animal`, etc.) Moreover, changing the inheritance hierarchy normally means changing the pattern of class and instance variables in the hierarchy — our new class, `Fish`, will contain information relevant to all its subclasses, and this information will have to be extracted from its previous position in the hierarchy and brought together under the new heading.
3. *Is the generative order adequately decoupled?* Not in Smalltalk-80, where inheritance hierarchies must be created top-down. We lack at present any published evidence on how Smalltalk programmers work, but experts have criticised this aspect in precisely the terms I would expect, namely that when designing the inheritance hierarchy the mental order in which steps are generated is not top-down, so that the effect of the environment’s insistence on top-down working is constrictive (Goldstein & Bobrow 1981; LaLonde 1987). LaLonde’s description is that the expert first designs a specific data type, by focussing on a useful set of operations, and then that data type is positioned in a logical hierarchy.
4. *Is it role-expressive?* Again, not adequately; although the purpose of object-oriented programming is to clarify relations between parts of programs, one finds that relationships between methods are obscure. The only browsers provided operate on

explicit inheritance or message-passing links in the code, but the real relationships lie much deeper. Frequently, issues of implementation dictate the structure of what should be logical relationships: thus, for technical reasons, Bag and Set are both subclasses of Collection, and Array is not a subclass of Dictionary (even though dictionaries are a way to generalise arrays) but of Set. (Examples from (LaLonde, 1987.) These relationships are not explicit and can hardly help the reader.

A different form of problem with role-expressiveness is this typical Smalltalk cliché:

angle
 ↑angle

This is a method, ‘angle’, owned by some object. The method refers to an instance variable, also called ‘angle’. The code means “if the object receives the *message* angle then return the value of the *instance variable* called ‘angle’.” Two different types of entity with the same name — and this potential confusion occurs once for every instance variable whose value can be accessed. When the identifier angle occurs in a program, which is it referring to? I have no evidence that this causes problems, but it certainly defies the requirements of cue validity.

5. *Are there any hard mental operations?* This is not really a question about the relation between the notation and the support environment, but it is just as important, or maybe more so. In the case of Smalltalk, the answer is resoundingly “Yes”, since work by O’Shea (1986) has demonstrated very definitely that the Smalltalk meta-class is a very severe learning obstacle. It can even be argued that the whole principle of abstraction, on which inheritance systems are based, is hard, and that object-oriented systems based on copies and prototypes (or exemplars) may be very considerably easier for average users to master.

9. Conclusions: the Concept of Dimensions

Having developed the ideas and demonstrated that they can be cashed, it is time to reconsider the basic notion. The physicists’ dimensions form a closed set. Do these cognitive dimensions form a closed set, or are they just plucked from the air and from experts’ insights, like guidelines? I suggest that the closest analogy may actually be with concepts taken from biology, such as plant growth. The mechanism of plant growth is extremely complex, but we know that normal green plants need phosphates, nitrogen, sunlight, etc., to grow successfully. We also know something about how plants respond to different combinations. Best of all, we understand a good deal about the mechanism of plant growth, so that we can make many predictive assertions. What plant biologists do *not* have, and cannot have, is a logically complete closure of orthogonal concepts, analogous to the physicists’ dimensions. There may always be another trace substance, undetected as yet, required for vigorous growth.

My intention is that these cognitive dimensions should be tied to a clear model of user activity. Given a particular model of behaviour, such as opportunistic planning, we can state a good deal about the requirements that must be met by the system (the combination of notation-plus-environment). To a limited degree we can go on to suggest how the model will behave if the those requirements are not met. A careful analysis of a model should yield a closed set of dimensions, but of course as models become more accurate new requirements will be found and the set of ‘dimensions’ will grow.

Although this enterprise is ambitious, a successful scheme of cognitive dimensions would bring many advantages. First, description of relevant features of designs would be more compact, less ambiguous. We can, in future, refer to a particular notation as being, say, ‘viscous’, rather than having to spell out the point in laborious English: “If you want to change something, it’s a lot of work to undo what you’ve already done and modify it”.

Second, the analysis turns out to clarify an ill-understood issue: how particular environmental features complement the notation. We should be able to find ourselves saying “If this notation is so viscous, we’d better provide an agenda system in the environment to help with change management”, or “This animation tool does not serve the users’ real needs”. We can also make better comparisons of trade-off issues between different notational features. The benefit to designers is obvious.

Third, it becomes clear that auxiliary notations have an important role to play, especially in programming tasks (informal notations, intermediate representations, program design languages, programming methodologies, etc.): they function by allowing the user to adopt the preferred cognitive strategy even in apparently adverse circumstances. A classic case of further-research-is-needed.

Acknowledgements

Thanks to Rachel Bellamy, David Gilmore, Steve Payne, Darrell Raymond, Pat Wright, Richard Young and an anonymous referee, who have all contributed ideas or helped me to make better sense of mine.

References

- R Brooks [1983], Towards a theory of the comprehension of computer programs, *International Journal of Man-Machine Studies* 18, 543-555.
- P S Brown and J D Gould [1987], An experimental study of people creating spreadsheets, *ACM Transactions on Office Information Systems* 5, 258-272.
- S Card, T A Moran and A Newell [1983], *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates; Hillsdale, NJ, USA.
- J M Carroll and M B Rosson [1985], Usability specifications as a tool in iterative development in *Advances in Human-Computer Interaction I*, H R Hartson, ed., Ablex.
- J M Carroll, J C Thomas and A Malhotra [1979], Clinical-experimental analysis of design problem solving, *Design Studies* 1, 84-92.
- F Détienne [1989], Program understanding and knowledge organization: the influence of acquired schemata, in *Psychological Foundations of Human-Computer Interaction*, P Falzon, J-M Hoc & Y Waern, ed., Springer-Verlag. (in press).
- L Flower and J R Hayes [1980], The dynamics of composing: making plans and juggling constraints, in *Cognitive Processes in Writing*, L W Gregg & E R Steinberg, ed.; Erlbaum.
- D J Gilmore and T R G Green [1984], Comprehension and recall of miniature programs, *International Journal of Man-Machine Studies* 21, 31-48.
- D J Gilmore and T R G Green [1988], Programming plans and programming expertise, *The Quarterly Journal of Experimental Psychology* 40A, 423-442;
- I Goldstein and D Bobrow [1981], An experimental description-based programming environment: four reports. Technical Report CSL-81-3; Xerox PARC, California.
- W D Gray and J R Anderson [1987], Change-episodes in coding: when and how do programmers change their code? in *Empirical Studies of Programmers: Second Workshop*, G Olson, E Soloway, & S Sheppard; ed., Ablex.
- T R G Green [1977], Conditional program statements and their comprehensibility to professional programmers, *Journal of Occupational Psychology* 50, 93-109.
- T R G Green [1982], Pictures of programs and other processes, or how to do things with lines, *Behaviour and Information Technology* 1, 3-36.
- T R G Green, R K E Bellamy and J M Parker [1987], Parsing-gnisrap: a model of device use, in *Empirical Studies of Programmers: Second Workshop*, F M Olsen, C Sheppard & E Soloway, ed., Ablex, Norwood, NJ.
- T R G Green and A J Comah [1984]. The programmer's torch, in *Proceedings of Interact'84 — First IFIP Conference on Human-Computer Interaction*, B Shackel, ed., Elsevier Science, Amsterdam.

- R Guindon, M Krasner and B Curtis [1988] Breakdowns and processes during the early activities of software design by professionals, in *Empirical Studies of Programmers: Second Workshop*, G Olson, E Soloway, and S Sheppard, ed., Ablex.
- H R Hartson and D Hix [1989], Toward empirically derived methodologies and tools for human-computer interface development, *International Journal of Man-Machine Studies*, (in press).
- R Hewson [1987], Typographic design: one practitioner's view of old meets new Xerox. PARC, Unpub. MS.
- J-M Hoc [1988], Towards effective computer aids to planning in computer programming in *Working with Computers: Theory versus Outcome*, G C van der Veer, T R G Green, J-M Hoc and D M Murray, ed., Academic Press, London.
- A Kohne and G Weber [1987], Struedi, a Lisp structure editor for novice programmers, in *Proceedings of Interact'87 - Second IFIP Conference on Human-Computer Interaction*, H J Bullinger and B Shackel, ed., Elsevier Science. Amsterdam.
- W R LaLonde [1987], Designing families of data types using exemplars. School of Computer Science, Carleton University, Ottawa, Report No. SCS-TR-108.
- D Norman [1986], Cognitive engineering, in *User Centred System Design*, D Norman and S Draper, ed., Lawrence Erlbaum Associates, New Jersey, 31-62.
- T O'Shea [1986], Why object-oriented systems are hard to learn, in *Proceedings of the OOPSLA'86 Conference*, ACM, New York.
- S Payne and T R G Green [1986], Task-action grammars: a model of the mental representation of task languages, *Human-Computer Interaction*, 2, 93-133.
- M Petre [1988], Issues governing the suitability of programming languages for programming experts, in *People and Computers IV*, D M Jones and R Winder, ed., Cambridge University Press.
- R S Rist [1986], Plans in programming: definition, demonstration, and development, in *Empirical Studies of Programmers*, E Soloway and S Iyengar, ed., Ablex.
- M B Rosson and S Alpert [1988], The cognitive consequences of object-oriented design, IBM Research Report, Yorktown Heights New York.
- J I A Siddiqi [1985], A model of program designer behaviour, in *People and Computers: Designing the User Interface*, P Johnson and S Cook, ed., Cambridge University Press.
- M E Sime, T R G Green and D J Guest [1977], Scope marking in computer conditionals: a psychological evaluation. *International Journal of Man-Machine Studies* 9, 107-118.
- E Soloway and K Ehrlich [1984], Empirical studies of programming knowledge, *IEEE Transactions in Software Engineering*. SE-10, 595-609
- E Soloway, J Pinto, S Letovsky, D Littman and R Lampert [1988], Designing documentation to compensate for delocalized plans. *Communications of the ACM* 31,1259-1267 .
- R Southall [1988], Visual structure and the transmission of meaning, in *Document Manipulation and Typography*, J C van Vliet, ed., Cambridge University Press.
- N A Streitz [1988], Writing is rewriting: a cognitive framework for computer-aided authoring. Paper delivered at 4th European Conf. on Cognitive Ergonomics, Cambridge.
- W Visser [1988], Towards modelling the activity of design: an observational study on a specification stage, INRIA Technical Report, Oulu. Finland, (to appear in the proceedings of the IFAC/IFIP/IEA/IFORS Conference on Man-Machine Systems).
- M Weiser and J Lyle [1986], Experiments on slicing-based debugging aids, in *Empirical Studies of Programmers*, E Soloway and S Iyengar, ed., Ablex.

- A D Whitefield[1985], *Constructing and Applying a Model of the User for Computer System Development*. University College, London, PhD Thesis.
- R M Young [1983], Surrogates and mappings: two kinds of conceptual mappings for interactive devices, in *Mental Models*, D Gentner and A L Stevens, ed., Erlbaum.
- R M Young, T R G Green and T. Simon [1989] Programmable user models for predictive evaluation of interface designs. In K. Bice and C. Lewis, eds., *Proceedings of CHI'89*. ACM, New York.