Green, T. R. G. (2000) Instructions and descriptions: some cognitive aspects of programming and similar activities. Invited paper, in Di Gesù, V., Levialdi, S. and Tarantino, L., (Eds.) *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000).* New York: ACM Press, pp21-28. Also available in pdf form via www.ndirect.co.uk/~thomas.green/workStuff/papers/

Instructions and Descriptions: some cognitive aspects of programming and similar activities

T. R. G. Green

Computer-Based Learning Unit
University of Leeds
Leeds LS9 2JT, UK

thomas.green@ndirect.co.uk

ABSTRACT

The Cognitive Dimensions framework outlined here is generalised broad-brush approach to usability evaluation for all types of information artifact, from programming languages through interactive systems to domestic devices. It also has promise of interfacing successfully with organisational and sociological analyses.

Keywords

Usability evaluation, cognitive dimensions, notations, telephone, Prolog, spreadsheet, cognitive psychology.

1. INTRODUCTION

We are living through a technological revolution, in which much research is necessarily dominated by immediate aims and short-term goals, and most research papers report some new accomplishment. The accomplishment may be useful but generalisations from one creation to another are very weak, unless the second is a direct descendant from the first. This paper is a contrast.

Science-based engineering rests on idealisations (capacitance, gravity). Physical or chemical theory describing these idealisations is combined with experience and craft knowledge to give useful and reliable outputs. Briefly, science-based engineering describes known phenomena in ways that are potentially useful, and that is the goal of the work I shall describe.

I shall address, not particular devices or interfaces, but the world of 'information artifacts' – systems and devices that store, display, and manipulate information. Research in this area partakes sometimes of science, sometimes of engineering, sometimes of product design, and indeed even sometimes of sociology. I have sympathies with all of those but my own work has lain mainly in the first two.

I shall sketch a framework in which idealisations of user activities are related to idealisations of the properties of information artifacts. Unlike, say, capacitance, let alone gravity, these idealisations are not the result of years of reflection by scientists of genius; they are most definitely preliminary efforts, crude and in need of polishing. Nevertheless, even as it stands, I claim that the framework meets to a fair extent its goal of describing phenomena In potentially useful ways.

The phenomena it describes are those where cognitive resources and limitations meet information structures. One potential use is to foresee ways in which a particular structure might fail to meet its users' hopes. More importantly, the framework can help us to learn from successes and mistakes, by offering descriptions at a level of abstraction that encourages generalization. With its aid one can see that the identical problems bedevil very different devices, such as the 'fossil' problem that occurs both in Unix systems and in simple home music sequencers, leading to 'mature disfluency' [11]; or one can see that similar design manoeuvres with identical consequences have occurred both in the design of word-processors and in the design of domestic central heating controllers. In short, unlike any other HCI technique I know of, it is explicitly concerned with the internal structure of information and how that structure affects usability.

I call this framework the ***cognitive dimensions of notations*** [9] because, although I am not entirely happy with any of the words – neither 'cognitive', nor 'dimension', nor 'notation' – that seems to be the best phrase going. And although I take credit for many of the ideas, I must make it clear that many others have collaborated, criticised, clarified, and contributed. Some will be mentioned in this paper but I owe thanks to all of them.

Of course, such a framework is too complex to describe properly in this talk. I shall do no more than sketch the main features and some brief illustrations. See [15] for a full description; see [18] for a detailed application to one domain, that of visual programming languages.

The approach is very general. A good usability method for assessing the information structures of phones should also be a good method for assessing programming environments, and vice versa (allowing, perhaps, some specialisations in each case). The cognitive dimensions framework succeeds reasonably well in that aim. I shall therefore first sketch the framework and then demonstrate its application to a typical

domestic telephone and to the design of a programming environment, two very different situations.

Finally I shall suggest how the approach might interface to approaches at a different level of analysis, such as Nardi's analysis of spreadsheet usage [21].

2. COGNITIVE DIMENSIONS

The cognitive dimensions of notations framework (CDs) is intended to provide a quick and approximate ('broad-brush') approach to usability analysis. I believe it is important to develop such approaches, ones that non-specialists will find easy to understand and easy to use. The framework therefore avoids any kind of detailed cognitive analysis, although it has a cognitive underpinning. There are very few new ideas in the approach. Most of the ideas will seem very familiar, in fact – but few of them have been extracted and named. It is the process of lexicalisation that makes it possible to generalise from one situation to another, to foresee that the success of difficulty attending a particular artifact may be shared by another artifact with a very different superficial appearance.

By spurning the issues of superficial appearance, this framework has to give up the hope of achieving detailed performance time predictions, such as KLM [5] offers. What it gains is both generality and depth: fundamental similarities can be discerned between very different devices.

The CDs are therefore presented as an ***analytical vocabulary*** for design discussion. Many of the dimensions reflect common usability factors that experienced designers might have noticed, but did not have a name for. Giving them a name allows designers to discuss these factors easily. Furthermore, there is no perfect user interface, notation, or representation; designers must make ***trade-offs***. A discussion vocabulary allows the trade-offs to be discussed and their consequences anticipated. It is not possible to create a design that has perfect characteristics in every dimension -- making improvements along one dimension often results in degradation along another.

The framework applies to ***information artifacts*** including household appliances, telephones, and novel interaction devices as well as conventional computer systems. Most information artifacts are probably important to several stakeholders, with different priorities and different criteria. Necessarily my outlook as a cognitive psychologist best fits me to discuss certain types of criteria, and I shall concentrate on ease of use by relatively experienced users.

2.1 The fundamental question

Every evaluation technique asks one fundamental question. In the CDs framework, that question is: are the users' intended activities adequately supported by the structure of the information artifact?

And as a supplementary: if not, what design manoeuvre would fix it, and what trade-offs would be entailed?

So the evaluation, in a nutshell, consists in classifying the intended activities, analysing the cognitive dimensions, and deciding whether the requirements of the activities are met.

2.2 Activities

No usability analysis can proceed far unless we have some idea of what the artifact will be used for; yet at the same time, there are good reasons to avoid highly detailed task analyses; such analyses are lengthy to construct, require specialised experience, and in the end do not capture the labile nature of everyday activities. Instead of using a detailed task analysis, the CDs framework just classifies 6 generic activities. A given artifact will support some of these activities better than others, and the analyst should decide which ones are required.

incrementation:	adding cards to a cardfile, formulas to a spreadsheet or statements to a program
transcription:	copying book details to an index card; converting a formula into spreadsheet or code terms
modification:	changing the index terms in a library catalogue; changing layout of a spreadsheet; modifying spreadsheet or program for a different problem
exploratory design:	sketching; design of typography, software, etc; other cases where the final product cannot be envisaged and has to be 'discovered'
searching:	hunting for a known target, such as where a function is called
exploratory understanding:	discovering structure or algorithm, or discovering the basis of classification

Each activity places different demands on the system. Nothing in what follows is good or bad except with reference to an activity that we wish to support.

2.3 The Components of Information Artifacts

Information artifacts, as described in this framework, have four components:

an interaction language or notation;
an environment for editing the notation;
a medium of interaction;
and possibly, two kinds of sub-devices.

2.3.1 Interaction languages and notations

The notation is what the user sees and edits: letters, musical notes, graphic elements in CAD. When the 'notation' is invisible, we shall deem it an interaction language, such as pressing buttons on a radio, dialling numbers on a telephone, clicking the mouse on a computer.

2.3.2 Editing environments

Different editors have different ontologies. Some word processors have commands operating on paragraphs, sentences, words and letters, while others only recognise individual characters. Some editors allow access to history, others deny it. Editors are usually the sub-devices that support the creation and management of abstractions – see below.

2.3.3 Medium of interaction

Typical media are paper, visual displays, and auditory displays. The important attributes are persistence/transience and constrained-order/free-order. Button presses are transient (unless a history record is available, as a helper sub-device); writing on paper is persistent. We shall say little about it here but it should be noted that some information structures are only successful for their intended purpose when used with a persistent medium; for example, writing even a short program using speech alone, without access to any persistent medium, is extremely difficult.

2.3.4 Sub-devices

Many devices and structures contain sub-devices. Two kinds are distinguished. ***Helper devices*** offer a new view, e.g. cross-referencers in programs, outline views in word processors. Helper devices are not always so formal, however: If the user typically writes notes on the backs of envelopes or sticks Post-It notes on the side of the screen, they should be regarded as helper devices, part of the system. The CDs framework is meant to encompass as much of the system as seems reasonable, and if Post-It notes are typically part of the system, they should form part of the analysis.

Redefinition devices allow the main notation to be changed. Macro recorders in contemporary word processors allow a sequence of commands to be replaced by a single command. Macros are a typical abstraction, and systems that allow abstractions to be created or modified always require a sub-device to work as an abstraction manager.

Sub-devices, whether helper devices or redefinition devices, often have their own notations or interaction languages that are separate from the main notation of the system, and an independent set of cognitive dimensions. The dimensions of these devices must be analysed separately. Thus, the macro recorder has different properties from the word-processor in which it is embedded.

A typical example: In a system with weak support for exploratory design, such as Pascal or C++, users will probably sketch out designs using a different information artifact with different properties (e.g. paper) as a helper device. Such 'twin-device' systems seem to invite 'improvement' to eliminate the need to use paper, or whatever, but it is possible that they work very well as they are and do not need to be 'improved': the two partners are perhaps blessed with complementary virtues.

2.4 Notational Dimensions

The main point of the CDs framework is to consider the notations or interaction languages and how well they support the intended activities, given the environment, medium, and possible sub-devices. We do this by considering a set of 'dimensions'. Each dimension describes an aspect of an information structure that is reasonably general. Furthermore, any pair of dimensions can be manipulated independently of each other, although typically a third dimension must be allowed to change (*pairwise independence*)[1]. In today's talk I shall only discuss a small number of dimensions, and do so very briefly. The full list is currently 13, and each needs far more space than can be given here.

None of these dimensions is evaluative when considered on its own. Evaluation must always take into account the activities to be supported.

[1] The notion of pairwise independence has caused difficulties to some readers. To illustrate, consider a given mass of an ideal gas, with a temperature, a volume, and a pressure. The volume can be changed independently of the pressure but the temperature must change as well; or the volume can be changed independently of the temperature, but the pressure must change accordingly. Similarly for any other pair of dimensions.

Viscosity: resistance to change.

A viscous system needs many user actions to accomplish one goal. Changing all headings to upper-case may need one action per heading. (Environments containing suitable abstractions can reduce viscosity.) We distinguish *repetition* viscosity, many actions of the same type, from *knock-on* viscosity, where further actions are required to restore consistency.

Visibility: ability to view components easily.

Systems that bury information in encapsulations reduce visibility. Since examples are important for problem-solving, such systems are to be deprecated for exploratory activities; likewise, if consistency of transcription is to be maintained, high visibility may be needed.

Premature commitment: constraints on the order of doing things.

Self-explanatory. Examples: being forced to declare identifiers too soon; choosing a search path down a decision tree; having to select your cutlery before you choose your food.

Hidden dependencies: important links between entities are not visible.

If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions. Examples: cells of spreadsheets; style definitions in Word; complex class hierarchies; HTML links.

Role-expressiveness: the purpose of an entity is readily inferred.

Role-expressive notations make it easy to discover why the programmer or composer has built the structure in a particular way; in other notations each entity looks much the same and discovering their relationships is difficult. Assessing role-expressiveness requires a reasonable conjecture about cognitive representations (see the Prolog analysis below) but does not require the analyst to develop his/her own cognitive model or analysis.

Error-proneness: the notation invites mistakes and the system gives little protection.

Enough is known about the cognitive psychology of slips and errors to predict that certain notations will invite them. Prevention (e.g. check digits, declarations of identifiers, etc) can redeem the problem.

Abstraction: types and availability of abstraction mechanisms.

Abstractions (redefinitions) change the underlying notation. Macros, data structures, global find-and-replace commands, quick-dial telephone codes, and word-processor styles are all abstractions. Some are persistent, some are transient.

Abstractions, if the user is allowed to modify them, always require an abstraction manager -- a redefinition sub-device. It will sometimes have its own notation and environment (e.g. the Word style sheet manager) but not always (for example, a class hierarchy can be built in a conventional text editor).

Systems that allow many abstractions are potentially difficult to learn.

2.5 Evaluating Systems wrt Activities

At last we are in a position to evaluate something in the CDs framework. Evaluation has two steps. The first is to decide what generic activities a system is desired to support. Each generic activity has its own requirements in terms of cognitive

dimensions, so the second step is to scrutinise the system and determine how it lies on each dimension. If the two profiles match, all is well. A tentative tabulation of the support required for each generic activity can be found in [15].

For example, *transcription* is very undemanding. No new information is being created so premature commitment is not a problem. Nothing is being altered, so viscosity is not a problem. On the other hand, to preserve consistency of treatment from instance to instance, visibility may be required.

Incrementation creates new information and sometimes there may be problems with premature commitment. The most demanding activity is *exploratory design*. A sizeable literature on the cognitive psychology of design has established that designers continually make changes at many levels, from detailed tweaks to fundamental rebuildings. Viscosity has to be as low as possible, premature commitment needs to be reduced, visibility must be high, and role-expressiveness – understanding what the entities do – must be high.

Beware a temptation. Do not readily believe that your system is purely incrementational. As information accumulates users are likely to wnt to re-order it – a modification activity, which is much more demanding. (Example: at first, young people put books or music CDs on their shelves in random order; later they impose a bit of a system; still later, they probably revise the system, as their tastes change or as their collection grows.)

2.6 Trade-offs

A virtue of this framework is that it illuminates design manoeuvres in which one dimension is traded against another. Although no proper analysis of trade-offs exists, we can point to certain relationships. One way to reduce viscosity is to introduce abstractions, but that will always require an abstraction manager in which to define the abstractions and some early commitment to choose which abstractions to define. The abstractions themselves may then become viscous, introduce hidden dependencies, etc. This topic needs much more research.

3. EXAMPLES

The following two examples demonstrate the framework's applicability to very different types of information artifacts. The first is the humble telephone; the second deals with Prolog. The telephone example shows that even an apparently simple device has an internal structure that dictates many usability aspects. The Prolog example shows that CDs quickly reveal (at least in hindsight) reasons why an extensive software project might have difficulties.

3.1 The Telephone

Consider first a familiar device, a telephone with a memory. The intention is not to reveal new insights into telephones, but to make it clear how the terms of the framework are deployed and how the procedure works. To make it concrete, we shall consider a commercial model, the British Telecom Duet 80, but the major features are not unusual. (An interactive web-based virtual telephone with similar features is available [13]). This telephone has 10 memory slots, numbered 0-9, which allow numbers to be stored and recalled. There is no internal memory for names attached to numbers, but the handset hosts a piece of card ruled into 10 compartments, labelled 0 to 9, on which the instruction book advises the user to record names associated with the memories (and furthermore, advises the use of a pencil, rather than a pen, to allow emendation).

The *main interaction language* is obviously the numbers used in dialling; they are expressed in a *transient* medium, as button presses, and the environment for manipulating the notation is nil – all you can do is to dial the number, it cannot be edited in any way.

There are two *sub-devices*. The first is the memory for storing codes. This is a 'redefinition' device, since it allows button sequences to be redefined as 'meaning' other button presses. The second sub-device, the piece of card recording the stored codes, is a helper device – it has no effect on the main notation, it just records what was stored in the memories.

For the main device, the principal user activity is *transcription*, from a written telephone number (in a directory, for instance) to a sequence of button presses. This is a very simple artifact. The interaction language is well-known to be error-prone (i.e. dialling errors are frequent, especially doublet errors such as turning 1134 into 1334) and highly viscous: in the environment of the BT Duet 80, as of many of domestic models, there is no Undo function, and after if any one of up to say 15 or more digits is misdialled, the user must cancel the attempt and start again. The user can choose whether or not to make use of the stored codes, so this device is abstraction-tolerant, but has to decide in advance which codes to store, so early, possibly premature, commitment is required.

For the redefinition device (the memory) deciding on the principal user activity is a bit harder: when the telephone is first used, the activity will be *incrementation* – storing the number for Aunt Mary, for example. Later, however, there could be other activities. You may have forgotten whether you stored the number for Aunt Mary, and want to find out – a *search* activity. Because the memories have no *visibility* at all, you will find it very difficult to know (unless you recorded all your stored codes using the 'helper device', also known as the piece of card).

When the memories are full, you may need to re-allocate them, because you have realised that it is more important to be able to dial the emergency services quickly than to dial Aunt Mary quickly. That is a *modification* activity. Individual memories can easily be reset – the *viscosity* is low, and each one can be changed independently of the others – there are *no hidden dependencies*.

Consider, however, the problem of keeping the numbers in a defined order. Perhaps you believe in putting the fire service number into the first slot, so that you can dial it readily, and so on. If you find yourself moving all the existing numbers into new positions, you will find that the telephone has very high viscosity.

Lastly, the remaining sub-device, the piece of card, so thoughtfully supplied with the device. This is principally used for *transcription* (e.g. recording the code for Aunt Mary), but must also be sometimes used for *modification* (when you decide to delete Aunt Mary in favour of the police). Transcription is not impaired by high viscosity (unless the notation is liable to errors – for now, we'll assume that it isn't) but modification obviously depends on low viscosity. You are therefore well advised to use pencil, not pen!

3.2 Textual vs. graphical Prolog editors

The logic programming language Prolog has much to offer, but is notoriously difficult for novices. In the hope of ameliorating their problems a tree-based graphical editor, Ted, was implemented but an evaluation [22] gave disappointing results. In a fine example of analysis-by-hindsight, I have argued [12] that for all its other virtues, Ted failed to solve some of the major usability problems of conventional Prolog. A précis of that analysis follows.

3.2.1 CDs of conventional Prolog

The CDs analysis of conventional Prolog was made on the assumption that for many Prolog novices, building even a small fragment of code is an exercise in exploratory design, as has been shown to be the case with novice users of other programming languages. The successful support of exploratory design places certain stringent demands on the notation and its working environment: low viscosity and high role-expressiveness are essential. These turn out to be problematic in Prolog. (Other CDs will not be discussed, for obvious space economics.)

Viscosity has to be considered for individual tasks, so a basket of representative tasks is necessary to create a representative overall measure. On the whole Prolog has little viscosity, even in a conventional text editor, but adding a new 'programming plan', though not difficult, has some unusual properties. A simple illustration, from Green [10], showed that for a plausible example, structured Basic has less viscosity than Prolog. (Siddiqi and Roast [26] offered a Prolog version that was less viscous than Green's, but even so, it was more viscous than the Basic program.) The example in question starts with a program to compute the average of a series of integers and modifies it to a program that also computes the 'filtered average' of the series, where the filtered average is the average of all non-zero elements; that is a reasonable example of the sort of modification a novice might make as the program was developed.

Role-expressiveness, the other important requirement, requires a little knowledge of the cognitive psychology of programming for its analysis. Assuming that programs are written by translating cognitive structures into code, it follows that part of the process of comprehending a program is parsing it back into the original cognitive components [14]. What will these cognitive structures be? Historically, it has been proposed that Pascal novices think in terms of 'plans', groups of statements that typically consume some data, operate upon on it in some characteristic way, and produce results for use by another plan [27] [29]; there is some empirical evidence to support this proposal, although the 'plan' is at best only one of the possible cognitive structures that programmers may employ. (E.g. Pennington [28] demonstrated that text structures were also relevant; Gilmore and Green [7] demonstrated that control-flow structures were also important for Pascal novices; Bellamy and Gilmore [1] suggest that the apparent structure depends on the task; Burkhardt et al., [4] distinguish between different types of text-derived structures, the textbase (from reading-to-recall) and the situation model (from reading-to-do); Green and Navarro [17] showed that visual/spatial components are sometimes relevant; and so on.). Practically the only study on Prolog [28] found evidence suggesting that a closely similar structure, the 'schema' [6] was a good fit to the recall data of Prolog experts and to a lesser extent of Prolog novices.

Role-expressiveness, in the context of programming, then becomes a measure of how easily the code can be parsed into 'plans' or 'schemas'.

Anecdotal evidence suggests that Prolog is much harder to decompose than Pascal: according to at least one theory of parsing, that is exactly what one would expect. Prolog, like assembly code, allows many different program structures to be built by combining a very small number of notational elements in different ways. There are no lexical distinctions between these elements, merely different rearrangements of the same symbols. Pascal, on the other hand, has plentiful lexical cues which help to distinguish different program structures from each other.

The importance of lexical cues in human natural language parsing was postulated many years ago [3], [8]. Pascal has many important lexical indicators, such as *while*, *for*, etc., which immediately distinguish between different syntactic constructs and which give big clues to distinguishing between different schemas; Prolog, in contrast, has very few. Compare the Pascal and Prolog versions of a count fragment (Figure 1). In each case we have some distinctive clues (the zero and the expression `N=N+1` or its Prolog equivalent) but the process that is being applied is more clearly indicated in the Pascal in two ways: the `while`, obviously enough, and the indentation, which is a strong help. Prolog, of course, has indentation, but it is less useful in picking out control structures.

```
[Pascal]
N := 0;
while expression true do
   begin
   N := N+1;
   compute new expression
   end

[Prolog]
count([],0).
count([A|B], N) :-
      count(B, N1),
      N is N1 + 1.
```

Figure 1: Pascal has lexical indications of control flow, Prolog does not.

(This sort of comparison is never clearcut. The most convenient data structure for Prolog is the list, which lends itself readily to algorithms that consume each member; lists are clumsy in Pascal, so I have used an abstract form of expression.)

Because of the lack of lexical cues, different Prolog programs can appear very similar on the page: subtle differences are important. The two fragments in Figure 2 use quite different techniques. To distinguish them it is necessary to see how the arguments are used in the body of the rule. (Don't think you can just count the arguments, because in real examples there will be other arguments dealing with other aspects of the processing.)

```
tripleA([], []).
tripleA([H1|T1], [H2|T2]):-
   H2 is 3*H1,
   tripleA(T1, T2).

tripleB([], Y, Y).
tripleB([H1|T1], X, Y):-
   H2 is 3*H1,
   tripleB(T1,[H2|X],Y).
```

Figure 2: two Prolog fragments appear similar but work differently.

How relevant is the lack of lexical indicators? If their absence makes it difficult to parse for relevant structures, then role-expressiveness will be impaired. Green and Borning [16] demonstrated just how poor Prolog's role-expressiveness might be. They applied a plausible and successful psycholinguistic model of human natural-language parsing [20] to the parsing of Prolog and Pascal, using the counting fragment as their simplest example. Obviously, they were not concerned with whether Prolog sentences were syntactically correct, but with extracting chunks corresponding to postulated cognitive structures, such as 'programming plans'.

In this model, there can be many fragments that are candidates for unification with each other. If the wrong ones join up, the parsing will fail. The more lexical cues are present, and the more powerful they are, the better the success rate. When applied to Prolog, with its relative dearth of lexical cues, the success rate was much less than when applied to equivalent programs in Pascal. Worse, when the complexity of the program was increased by adding extra components, the difficulty of parsing the Prolog version increased much faster than the difficulty of parsing the Pascal version.

3.2.1.1 CDs of the graphical editor Ted

The graphical editor displayed a tree of 'techniques', useful combinations of Prolog code [2]. A program was developed by adding a new technique to the tree. The displayed showed the current state of the Prolog code and the associated tree. Deleting material removed it completely – no access to past history was possible.

Viscosity: Programmers using Ted had only one way to revise their code: they had to edit the history tree. To change the program, learners had to step back through the tree to where the offending component had been added, and then delete that component, *thereby at the same deleting every subsequent editing operation.*

Evidently the knock-on viscosity could be extremely high if the learner needed to change a choice that had been made far back in the tree. To avoid this fate (premature commitment), learners no doubt attempted to look far ahead, which may or may not have been pedagogically useful.

It would seem that the Ted designers thought of the coding activity as *incrementation*, rather than as *exploratory design*; an easy mistake, but with serious consequences. This is the temptation mentioned above.

Role-expressiveness: Since the Ted editor displayed standard Prolog as its output code, the role-expressiveness was not improved. The role-expressiveness of the techniques was also very poor, because the output code does not indicate which part of the code is associated with which technique.

Thus, overall the role-expressiveness might been made *worse*, not better. (It would be possible to make an empirical test of role-expressiveness, which would confirm or refute my conjecture.)

In conventional Prolog, the opportunities for *secondary notation* are severely limited – comments and indenting, nothing more – but it seems that in Ted, even those possibilities were removed: learners were therefore unable to document the reasons for their editing choices.

On the other hand, the Ted editor reduced some of the *error-proneness* of conventional Prolog, by prebventing mismatched parentheses, inconsistent spelling, unfinished comments, and so on. Trivial though they may be, these add to learners' problems and in the case of inconsistent spelling they can be very difficult to catch.

3.2.2 Textual vs graphical Prolog: conclusions

If my cursory analysis is correct (and it must be emphasized again that it *is* cursory, and is not based on experience with the actual tool), then Ted seems to have taken a wrong path. The problems that were identified in conventional Prolog, using the cognitive dimensions framework, were poor role-expressiveness and high viscosity. Ted seems to have made no contribution to solving those problems.

With hindsight, therefore, it seems not too surprising that the results were disappointing. Since the Ted environment had many excellent features, it would be good to see further work in the same direction.

4. INTERFACING TO OTHER LEVELS OF ANALYSIS

It is now well understood that information artifacts exist in a social or organisational situation. 'Mere' usability is not enough; they have to fit into patterns of activity. Yet few people can be specialists in more than one area, and it is regrettably the case that usability analyses and sociological or related analyses often pass each other by.

I would therefore like to finish this sketch of the Cognitive Dimensions framework by describing what I believe is a successful link-up, in the area of that very popular artifact, the spreadsheet. Nardi [21] has reported studies of spreadsheet use in organisations and describes programming communities of co-operating users. She distinguished three major roles: end-users, supported by the strong visual formalism of the spreadsheet, who found the formula language relatively easy because it contained many domain-specific functions; a mid-level group, who had no more training than end-users but who adopted the role of crating small solutions for local needs; and specialists at the system level. She also found that the spreadsheet was an effective communication medium across different levels and specialities in the organisation.

Hendry and Green [19] made a further study, aimed not at the organisational aspects but at the individual usability, yet using Nardi's interview-based methods. We found that the individual usability reports confirmed much of Nardi's analysis. We also reported on the cognition dimensions approach to the spreadsheet. It emerged as an excellent device for incremental usage, so that adding more formulas to an existing spreadsheet was very easy. On the other hand, the absence of any abstraction mechanisms, the poor role-expressiveness and the pervasive hidden dependencies encouraged undetected errors and made the inner workings of

any spreadsheet hard to grasp (poor exploratory understanding) – although if the programmer (or anyone else who has the workings in their head) is present, the lack of abstraction facilities makes it very easy to explain what the spreadsheet does, much easier than explaining an OO program. Hence, the spreadsheet becomes an effective communication device.

For similar reasons, the CDs analysis suggests that exploratory design would be by no means as easy as incrementation, which was confirmed by the reports from our informants. Thus, one can readily see that these qualities would be likely to encourage the separation of a pure end-user role from a mid-level role.

There is more to be said about spreadsheets, but my purpose is not to present all that is known, but to demonstrate that the cognitive dimensions analysis can link up to an organisational-level analysis, explaining why an artifact comes to be used in particular ways. This is an area where it would be extremely interesting to see far more research.

5. ACKNOWLEDGEMENTS

The framework has benefited from discussions with many colleagues, especially Alan Blackwell, Rachel Bellamy, David Gilmore and Marian Petre, but also many others. My thanks to all.

6. REFERENCES

[1] Bellamy, R. K. E. and Gilmore, D. J. (1990) Programming plans: internal or external structures? In K. J. Gilhooly, M. T. G. Keane, R. H. Logie and G. Erdos (Eds.) *Lines of Thinking: Reflections on the Psychology of Thought*. (Vol 1.) Wiley. 59-71

[2] Bowles, A. and Brna, P. (1999). Introductory Prolog: A suitable selection of programming techniques. In P. Brna, B. du Boulay and H. Pain (Eds.), *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Stamford, CT: Ablex

[3] Bratley, P., Dewar, H. and Thorne, J. P. (1967) Recognition of syntactic structure by computer. *Nature*, Vol. 216, No. 5119, 969-973.

[4] Burkhardt, J.-M., Détienne, F. and Wiedenbeck, S. (1997) Mental representations constructed by experts and novices in object-oriented program comprehension. In Howard S., Hammond J. and Lindgaard J. (Eds) *INTERACT '97*. Sydney: Chapman & Hall. 339-346

[5] Card, S. K., Moran, T. P. and Newell, A. (1983) *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum.

[6] Gegg-Harrison, T. S. (1991) Learning Prolog in a schema-based environment. *Instructional Science*, 20, 173-192.

[7] Gilmore, D. J. and Green, T. R. G. (1988) Programming plans and programming expertise. *Quarterly J. Exp. Psychol*. 40A, 423-442.

[8] 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.

[9] Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge: Cambridge University Press. 443-460

[10] Green, T. R. G. (1990) The cognitive dimension of viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds) *INTERACT '90*. Amsterdam: Elsevier. pp 79-86

[11] Green, T. R. G. (1995) Looking through HCI. Invited paper at 10th Annual Workshop of BCS HCI Group. In Kirby, M. A. R., Dix, A. J. and Finlay, J. E. (Eds.) *People and Computers X*. CUP

[12] Green, T. R. G. Building and manipulating complex information structures: issues in Prolog programming. In P. Brna, B. du Boulay and H. Pain (Eds.), *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Stamford, CT: Ablex, 1999. (Chapter 1, pp. 7 –27). See http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/PrologChap.RTF

[13] Green, T. R. G. (1999) A virtual telephone. Web-based document, URL: http://www.ndirect.co.uk/~thomas.green/workStuff/devices/phones/SmartPhoneE.html

[14] Green, T. R. G., Bellamy, R. K. E. & Parker, J. M. (1987). Parsing and Gnisrap: a model of device use. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex. 132-146

[15] Green, T. R. G. and Blackwell, A. F. (1998) Cognitive dimensions of information artifacts: a tutorial. Web-based document. URL: `http://www.cl.cam.ac.uk/users/afb21/publications/CDtutSep98.pdf`

[16] Green, T. R. G. and Borning, A. (1990) The Generalized Unification Parser: modelling the parsing of notations. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) *INTERACT '90*. Amsterdam: Elsevier. 951-957.

[17] Green, T. R. G. and Navarro, R. (1995) Programming plans, imagery, and visual programming. In Nordby, K., Helmersen, P. H., Gilmore, D. J, and Arnesen, S. (1995) *INTERACT-95*. London: Chapman and Hall (pp. 139-144).

[18] Green, T. R. G. and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7, 131-174.

[19] Hendry, D. G. and Green, T. R. G. (1994) Creating, comprehending, and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *Int. J. Human-Computer Studies*, 40(6), 1033-1065.

[20] Kempen, G. and Vosse, T. (1989) Incremental syntactic tree formation in human sentence processing: an interactive architecture based on activation decay and simulated annealing. *Connection Science*, 1, 273-290.

[21] Nardi, B. (1993) *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press.

[22] Ormerod, T. C. and Ball, L. J. An empirical evaluation of Ted, a techniques editor. for Prolog programming. In W. D. Gray and D. A. Boehm-Davis (Eds.), *Empirical Studies*

of Programmers: Sixth Workshop. Newark, N. J. : Ablex, 1996.

[23] Pennington, N. (1987) Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 295-341.

[24] Rist, R. S. (1986) Plans in programming: definition, demonstration, and development. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*. Norwood, NJ: Ablex. 28-47.

[25] Romero, P. (1999). Focal structures in Prolog. In Green T., and Brna P. (Eds.), *Proceedings of the Psychology of Programming Interest Group*, 11th Workshop. http://www.cogs.susx.ac.uk/users/juanr/elicit.ps

[26] Siddiqi, J. I. and Roast, C. R. (1997) Viscosity as a metaphor for measuring modifiability. *IEE Proc. Software Engineering*, 144(4: August), 215-223.

[27] Soloway, E. and Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, 595-6