Notational Systems – the Cognitive Dimensions of Notations framework

Alan F. Blackwell and Thomas R.G. Green

Final version of manuscript as included in
John M. Carroll (Ed.)
HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science
San Francisco: Morgan Kaufmann (2003), pp. 103-134.

1. Motivation

The field of HCI is gradually achieving sound theoretical descriptions of the activities, context and cognition of computer system users. How do these research results get applied by the people who design new user interfaces? Although we have theoretical descriptions of the activities of system users, we have fewer descriptions of the design activities of user interface designers. There are certainly theory-based design methods prescribing the things that designers ought to do. Almost all of these have been developed from the perspective of descriptions of the user, rather than from consideration of the needs of designers – the current vogue for "user-centred design" clearly expresses this emphasis on the user. According to this perspective, if user interface designers are to apply research into user needs, those designers must be able (and interested) to interpret and apply theoretical results.

Alternatively, there are also many popular approaches to user interface design that have minimal theoretical grounding. Such design methodologies generally attempt to present designers with a checklist (Nielsen & Molich 1990) or a procedural list of design activities (Wharton, Rieman et. al. 1994) that will generate a good design. The reduction of design to a checklist or a predefined procedure is widely proposed in other areas of software development, not just user interface design. Popular software engineering texts provide a series of checklists that aim to assure design quality at different stages of a project (Pressman 1997), while structured and object-oriented design methods prescribe a sequence of design tasks that will generate software from requirements (Booch, Rumbaugh & Jacobson 1999).

Although common in software engineering, few other design disciplines proceed according to checklists or predefined sequences of tasks. Software design educators increasingly suggest that such design tools are not sufficient for the design of novel user interfaces either (Winograd 1996). The highly structured (but largely a-theoretical) checklist methods have worked reasonably well in the development of user interfaces built from a small set of standard elements, such as Motif widgets, the Macintosh toolbox, or Windows foundation classes. This is because the design choices in arranging menus, buttons and dialog boxes are relatively few, and the results can sometimes be assessed using checklists such as those presented in "style guides" for the relevant system (Apple 1992, Microsoft 1995). Simple checklists are not so successful when the user interface is composed of completely novel graphical elements, composed according to completely novel visual grammars. These notational systems range from complex products such as visual programming languages, to embedded functionality such as central heating controls and the ubiquitous computing platforms of the future that move beyond the windows, icon and menu paradigm.

We do not believe that it will be possible to deal with these new notational systems by creating new checklists. Instead, we emulate other design disciplines by aiming to support the activity of the designers, based on some understanding of the process of design. User interface design is currently far more of a craft than an engineering discipline. It is subject to elements of affect, of fashion and social acceptance, in addition to technical considerations. For these reasons, we can learn from studies of other design disciplines where the same craft elements apply. For example, a study comparing knitwear designers and helicopter designers (Eckert & Stacey 2000) observes that a community of designers develops their own vocabulary for design criteria that is created through practice and tradition, rather than being easily accessible outside the community. In the Cognitive Dimensions of Notations framework, we aim to provide the same kind of vocabulary for use by user interface designers.

1.1. Example

Back in the 80's one of us (TG) was shown a system that he was sure wouldn't work. Yet the received wisdom of 80's HCI seemed to have no explanation of the problems he foresaw. It was a speech-to-text dictation system for use with the Pascal programming language. The speech-recognition technology appeared to be entirely adequate for this restricted language. What problem could there be?

What was wrong with that system was an unfortunate combination. First, it was intended for users doing a kind of design activity; second, the language contained many internal dependencies, and third, it severely restricted the user's order of actions. The system worked fine when a document was dictated to it. But real users do not design like that. Design research shows that they continually change things. And they do not start at the beginning, but rather in the middle. This system would not allow such behaviour.

It is easy to experience the problem. Try dictating a program in Pascal or even Java. The program must be dictated in proper order from start to finish. *It cannot be written down in advance*, because this system was intended for users who were unable to use their hands – suffering from severe arthritis, for example. Correctly dictating even a short program straight from the head is nearly impossible, and the editing commands provided by this system made the problem even harder, because they required the program to remain syntactically valid at all times.

This case made it apparent that existing HCI had no way to describe the structure of information, and therefore had no way to analyse the interactions between: a) the structure of the information, b) the environment that allowed that structure to be manipulated, and c) the type of activity the user wanted to perform. In short, the system forced programmers to make decisions before they had the information they needed to base the decisions on. We call this *premature commitment* (definition: constraints on the order of doing things). Moreover, changing those decisions was not easy. We call this second problem *viscosity* (definition: resistance to change). These are two examples of the cognitive dimensions of the notational system. In the rest of this chapter we introduce and analyse many others. Brief definitions are given as each term is introduced – more complete descriptions can be found later.

Once dimensions like these have been identified, it is easy to find other cases where they apply. We have seen a prototype conferencing system where, in order to make a contribution to a discussion, a participant had to start by selecting the type of contribution - a query, a rebuttal, or whatever. This is premature commitment again, for how does one know what one is saying before one says it? Maybe the user can look at the draft contribution and decide that on balance it's a query, not a rebuttal, and recategorise it; or maybe the system makes life still harder by making it impossible to recategorise without dumping the text written so far and starting over. (We've all used systems like that.) That's viscosity again. We can easily find these similar examples because the description of the problem is short and high-level, not a heavily-detailed analysis, and because it ignores specifics of representation in favour of the underlying structure of information.

HCI has a gap in that area. It has generated several sophisticated approaches that each address one type of user activity, such as routine cognitive skills or reasoning or menu-following. Within their scope these approaches can offer suggestions for redesign, but they usually focus on representation and they require extensive, detailed modelling. Some other kinds of user activity, notably design activity, have been studied intensively, so that we know much about the pattern of activity, but that observational work has not generated much to help design systems for design. But there is no approach that addresses all types of activity, that can lead to constructive suggestions for improving the system or device, that avoids details, and that allows similar problems to be readily identified in very different situations. The CDs framework fills the gap. It describes necessary (though not sufficient) conditions for usability, deriving usability predictions from the structural properties of a notation, the properties and resources of an environment, and the type of activity.

The motivation was thus:

- to offer a comprehensible, broad-brush evaluation (no 'death by detail');
- to use terms that were readily comprehended by non-specialists;

- to be applicable not just to interactive devices, which are the usual meat of HCI, but also to paper-based notations and other non-interactive information systems;
- to be theoretically coherent;
- and, especially, to distinguish between the needs of different types of user needs (such as the difference between dictation tasks and design tasks).

Furthermore, a CDs analysis frequently reveals a variety of interesting design choices. Although its broad-brush nature precludes detailed predictions, the framework can prompt designers to notice effects of their choices that had previously gone unseen. Perhaps most interestingly, the framework describes trade-offs between choices, showing how solving one type of user difficulty may create a different type.

2. Overview

The cognitive dimensions framework is not an analytic method. Rather, it is a set of discussion tools for use by designers and people evaluating designs.

The aim is to improve the quality of discussion. Experts make sophisticated judgements about systems, but they have difficulty talking about their judgements because they don't have a shared set of terms. Also, experts tend to make narrow judgements, based on their own needs of the moment and their guesses about what other people may need; and other experts don't always point out the omissions. Again, if they had a shared set of terms, and that set was fairly complete, it would prompt a more complete consideration.

In short, experts would be in a good position to make useful early judgements, if (i) they had better terms with which to think about the issues and discuss them, and (ii) there was some kind of reminder of issues to be considered. The terms might or might not describe a new idea; most probably, the expert will recognise a concept as something that had been in his or her mind, but had never before been clearly articulated and named.

Discussion tools are good concepts, not too detailed and not too woolly, that capture enough important aspects of something to make it much easier to talk about that thing. They promote discussion and informed evaluation.

To be effective, they must be *shared* - you and I must have the same vocabulary if we are going to talk. And it is better still if we share some *standard examples*. And it is best of all if we know some of the pros and cons - the *trade-offs* between one solution approach and another.

Discussion tools have many virtues: they elucidate notions that are vaguely known but unformulated; they prompt a higher level of discourse; they create goals and aspirations; they encourage re-use of ideas in new contexts; they give a basis for informed critique; they supply standard examples that become common currency; and they allow the inter-relationships of concepts to be appreciated.

Figure 1 illustrates a real-life discussion without the benefit of discussion tools; Figure 2 shows how it might have been if the participants had possessed shared concepts - shorter, more accurate, and less frustrating. In this version of the discussion, a number of new terms have been introduced, taken from the CDs framework – these are italicised.

A: ALL files in the book should be identical in everything except body pages. Master pages, paragraph formats, reference pages, should be the same.

B: Framemaker does provide this ... File -> Use Formats allows you to copy all or some formatting categories to all or some files in the book.

A: Grrrrrrrr .... Oh People Of Little Imagination !!!!!!

Sure I can do this ... manually, every time I change a reference page, master page, or paragraph format .....

What I was talking about was some mechanism that automatically detected when I had made such a change. (.....) Or better yet, putting all of these pages in a central database for the entire book ......

C: There is an argument against basing one paragraph style on another, a method several systems use. A change in a parent style may cause unexpected problems among the children. I have had some unpleasant surprises of this sort in Microsoft Word.

Figure 1 An impoverished discussion that would have been better if the discussants had shared appropriate concepts. This is a verbatim transcript from an actual newsgroup discussion, following an irritated message (by A) about how much work he had to do to keep identical formats for all components of a large project (a 'book' in the version of Framemaker that had the limitation being discussed).

A: I'm doing ***modification*** activities. I find Framemaker is too ***viscous***.

B: With respect to what task?

A: With respect to updating components of a book. It needs to have a higher ***abstraction level***, such as a style tree.

C: Watch out for the ***hidden dependencies*** of a style tree. Also, the abstraction level will be difficult to master; getting the styles right may ***impose lookahead***. Is the ***trade-off*** worth it?

Figure 2 An improved discussion (names of Cognitive Dimensions are emphasised)

There is a relatively well-defined series of steps to be taken by designers when applying Cognitive Dimensions analysis to a system design.

- step 1: get to know your system.
- step 2: decide what the user will be doing with the notation.
- step 3: choose some representative tasks.
- step 4: for each step in each task, askwhether the user can choose where to start; how a mistake will be corrected; what if there are second thoughts; what abstractions are being used; and so on, for the other dimensions. This will generate an observed profile.
- step 5: compare the observed profile with the ideal profile for that type of activity.

The formal apparatus consists of

- a list of generic types of notation use activities: in the Framemaker example, the activity is modifying the structure of the notated information.
- a list of dimensions: in the Framemaker example, one such dimension is 'viscosity', which means resistance to change
- a 'profile' for each activity: in the Framemaker example, the activity is modification of structure, which demands low viscosity (among other requirements)
- examples of trade-offs and design maneuvers, to warn of potential problems: in the Framemaker example, the proposed redesign would lower the viscosity but could readily introduce problems of hidden dependencies, etc.

What you get out of this approach is a rough and sketchy evaluation. As we saw back in Figure 1, it will correspond to what users talk about. And if you were to consider changing the design, it will alert you to some of the possible trade-off consequences.

An alternative approach to the application of Cognitive Dimensions is for system users to be given the dimensions as a framework in which to express their experience of using the system.

In this case, the nature of the expertise is different. The users of the system may not be expert in the possible design solutions, or in the space of trade-offs, but they do have expertise in the nature of interaction with the system. For this population, Cognitive Dimensions can provide a vocabulary in which to express the structure of their experience. They can also provide some insight into the possibility of design alternatives, even though the user may never have experienced any system other than the one they are familiar with.

It is important to note that the CDs framework is more than a checklist. The list of dimensions (which is included later in this chapter) may loom large in the mind of the reader, because there are quite a lot of them, and because some of them have an immediate intuitive appeal. Nevertheless, the list of dimensions is not sufficient on its own to achieve meaningful analysis of a notational system. It is essential to distinguish between different activities and different notations within the system, and to recognize that dimensions are not good or bad in themselves – they simply describe properties of the system with respect to those activities. In our experience, readers coming to the CDs framework for the first time are tempted to regard the dimensions simply as a checklist, applicable in the same way as Nielsen's list of evaluation heuristics. However it is important to understand the rest of the framework in order to start work.

3. Scientific foundations

There has been a long tradition in computer science of improving system usability by creating new design notations. Early programming languages such as FORTRAN and COBOL were intended to be reasonably familiar to the users (mathematicians and business people respectively), while describing system behaviour at a relatively high level of abstraction (compared to assembly language programming). Furthermore many innovations in software design have been accompanied by diagrammatic notations used to express the design. These extend from the earliest flowcharts to current development of the Unified Modeling Language (Booch, Rumbaugh & Jacobson 1999).

Where graphical editors are used to create, manipulate, and catalogue these design diagrams, it is natural to wonder whether the rather mechanical process of generating source code to implement the diagram could not also be automated. The result would be a programming language in which the design diagram was the program – a visual programming language. This has been an attractive goal for over 25 years. There is a research journal and a conference series devoted to visual programming languages – one survey of the field is that by Myers (1986). Some visual programming languages have become commercially successful – a report of user experiences with the LabVIEW language has been published by Whitley and Blackwell (2001).

Unfortunately, much research into visual programming languages was not informed by relevant models of human-computer interaction. A survey of the published literature (Blackwell 1996) revealed that much of it justified the cognitive advantages of visual languages in terms of concepts from popular psychology such as the left-brain right-brain dichotomy. This was in spite of published research into the usability implications of diagrammatic notations, dating back to the 1970s (Fitter & Green 1979). Empirical investigations of notation usability have tended to find that while diagrams are useful in some situations, they are not efficient in others (e.g. Green & Petre 1992). This is in striking contrast to the occasional claims that visual programming languages would be a panacea, providing a new generation of universally accessible programming tools.

Green Petre and Bellamy (1991) use the word 'superlativism' for the belief that there exists some notation that is universally best in its domain, whatever the activity may be. The converse view, which is surely correct (and has been empirically demonstrated in several situations) is *that every notation highlights some kinds of information, at the cost of obscuring other kinds*. Cognitive science and artificial intelligence research has often emphasised the importance of choice of representation in problem solving ("well represented is half solved"). As a result, the effect of notations and external representations on reasoning continues to be a popular research topic. One major stream of this work is in the theory and application of diagrams.

All visual notations have diagrammatic properties – even text has much in common with diagrams. A page like this one contains a great deal of linguistic structure, but it is not purely linguistic. Paragraphs are separated into visual units to indicate thematic structure; page numbers and citations provide overlaid reference schemes; variations in font exploit visual pop-out effects for local emphasis, chapter headings and marginalia offer context and commentary. The textual representation of spoken language has undergone radical improvements in notational usability over the last two millenia, for example the discovery in the 15th century that using of white space to separate words (BEFORETHATTHEYWROTEWITHOUTSPACES), enabled scholars to read without speaking aloud (Robinson 1995). It is not only text that has a surprising degree of diagrammatic content. Even "representational" paintings and photographs include a great deal of diagrammatic structure -- compositional elements, conventional symbolisation, or meaningful cropped borders (Kress & van Leeuwen 1996).

Diagrams research generally falls into two classes -- detailed explanations of specific properties of notations, and analyses of cognitive tasks employing notations. A good example of the former is Shimojima's formal description of *Free Rides*: in which new information is created simply by following syntactic conventions (Shimojima 1996). Cheng has described the educational use of Law Encoding Diagrams, in which free rides in the form of geometric relationships express valid conclusions regarding mathematical or physical laws (Cheng 1998). These analyses give detailed insight into one mechanism for the Cognitive Dimension of *Closeness of Mapping* (definition: closeness of the representation to the domain). One example of a cognitive task employing notations is that of creative ambiguity, described by researchers including Fish & Scrivener (1990), Goldschmidt (1991), and Suwa & Tversky (1997). They describe the process by which designers return to their sketches for new ideas. In order to allow this, the sketches have been left deliberately incomplete or ambiguous. This is an example of one application of the Cognitive Dimension of *Provisionality* (definition: degree of commitment to actions or marks).

The CDs framework does not consider representational issues in themselves: the framework is solely based on structural properties. Questions both of effectiveness (e.g. optimum size of buttons) and of aesthetics are outside its purview. This restriction has two great advantages. The first is that identical (or rather, isomorphic) problems can be spotted even when they occur in very different guises. An early example (from Green 1989) observed that the file structure of a computer operating system has much in common with the structure of a pattern sequencer for a drum machine, and therefore that problems and solutions in the one case would probably apply to the other.

The second great advantage of avoiding representational issues is that the CDs framework is thereby applicable to systems long before they reach the stage of being prototyped. Any technique that uses Fitts' Law, such as the Keystroke Level Model, or that considers 'whether the user's language is used', such as Heuristic Evaluation, is solely applicable to designs that have at least reached detailed specification.

Despite being applicable to all types of information artifacts, this framework has come to prominence, naturally enough, in an area where conventional HCI techniques have little clear applicability and where a good deal of design activity has been going on – namely, in visual programming languages and environments. Here, the typical user activity is non-routine design; the notations (programming language) vary considerably; and the features of the environments also vary considerably: Green & Petre 1995, Green 1999, Burnett, Cao and Atwood 2000.

As we stated in the introduction, we believe that user interface design, and notation design, are engineering design crafts, not sciences. Like any other design crafts, they combine elements of science with many other elements: peer-assessed best practice, market pull, fashion, trial-and-error, aesthetics, individual flair, and whatever else contributes to design. The cognitive dimensions framework only addresses some aspects of this *melange*. The framework is not the result of importing and specialising any single viewpoint from another science. Rather, it is a synthesis from several sources, and because we have much to learn in this area, it constitutes a patchwork or a quilt with holes where suitable pieces could not yet be found.

4. Detailed description

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. Note that the term "activity" is used differently in this context than other chapters of this book. A notational activity in cognitive dimensions is a manner of interacting with a notation. This use of the word is unrelated to Activity Theory, or Active User Theory.

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 contrasts 6 generic types of notation use activity. Note that this list was not derived from a single theoretical precursor, but has been extended and refined through contributions from a variety of researchers (e.g. Blackwell, Britton et. al. 2001) since the framework was first proposed. It is assumed that 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.

The Components of Notational Systems

Notational systems, 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*.

Interaction languages and notations

The notation is what the user sees and edits: letters, musical notes, graphic elements in CAD.

Editing environments

Different editors have different assumptions. 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.

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 view is available); writing on paper is persistent. 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.

Sub-devices

Many devices and structures contain sub-devices. Two kinds are distinguished.

Helper devices offer a new notational view, e.g. cross-referencers in programs, outline views in word processors, a history view of button presses. 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.

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].

Here we can only discuss a small number of dimensions, and do so very briefly. The full list is currently 13 (although several more are proposed, as discussed later), and each needs far more space than can be given here. A proper presentation for reference purposes should include, besides the brief definition, a thumbnail illustration; an explanation; an account of its cognitive relevance; an estimate of its cost implications; an account of subtypes and examples; and a list of work-arounds and remedies, with their associated trade-offs.

IMPORTANT: 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. There are sometimes actions that cause dependencies to get frozen – e.g. soft figure numbering can be frozen when changing platforms; these interactions with changes over time are still problematic in the framework.

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

Role-expressive notations make it easy to discover why the author 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.

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.

Secondary notation: extra information in means other than formal syntax.

Users often need to record things that have not been anticipated by the notation designer. Rather than anticipating every possible user requirement, many systems support secondary notations that can be used however the user likes. One example is comments in a programming language, another is the use of colours or format choices to indicate information additional to the content of text.

Closeness of mapping: closeness of representation to domain.

How closely related is the notation to the result it is describing?

Consistency: similar semantics are expressed in similar syntactic forms.

Users often infer the structure of information artefacts from patterns in notation. If similar information is obscured by presenting it in different ways, usability is compromised.

Diffuseness: verbosity of language.

Some notations can be annoyingly long-winded, or occupy too much valuable "real-estate" within a display area. Big icons and long words reduce the available working area.

Hard mental operations: high demand on cognitive resources.

A notation can make things complex or difficult to work out in your head, by making inordinate demands on working memory, or requiring deeply nested goal structures.

Provisionality: degree of commitment to actions or marks.

Even if there are hard constraints on the order of doing things (premature commitment), it can be useful to make provisional actions such as recording potential design options, sketching, or playing "what-if" games. Not all notational systems allow users to fool around or make sketchy markings.

Progressive evaluation: work-to-date can be checked at any time.

Evaluation is an important part of a design process, and notational systems can facilitate evaluation by allowing users to stop in the middle to check work so far, find out how much progress has been made, or check what stage in the work they are up to. A major advantage of interpreted programming environments such as BASIC is that users can try out partially-completed versions of the product program, perhaps leaving type information or declarations incomplete.

Profiles

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 Green and Blackwell (1998).

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.

Trade-offs

A virtue of this framework is that it illuminates design maneuvers 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, but some of the relationships we have observed are exhibited in Figure 3.

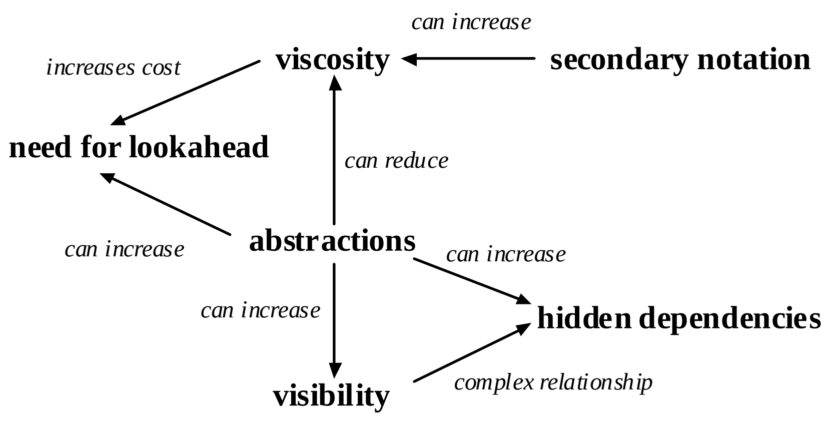

Figure 3 - some typical trade-offs

Use by an analyst

As a usability evaluation technique becomes widespread, its popularity depends more on accessibility and ease of use for the evaluator, rather than the original scientific origins of the technique. The most widely used techniques offer an extensive support apparatus of training and educational resources, analysis tools, review and tutorial literature. The support infrastructure for CDs has developed from a variety of invited lectures and tutorial sessions at conferences and trade meetings, to a comprehensive written tutorial (Green and Blackwell 1998), now distributed freely from a CDs resources site (URLs are given at the end of this chapter)This written tutorial complements extensive descriptions of each dimension with a strong example, descriptions of the consequences that may arise from problems in that dimension, potential workarounds, likely tradeoffs with other dimensions, some design maneuvers and so on. Many aspects of the dimensions are illustrated in simple interactive web-based devices showing differing designs for the same problem, so that readers can experiment with and experience the usability consequences of the dimensions. These tools allow new users of the framework easily to develop the required approach to analysis.

A questionnaire approach

In the context of usability studies, CDs have also been used to structure user feedback questionnaires. The fact that the dimensions describe generic aspects of usability, rather than features of the system under study, leaves the user free to comment on aspects of the system that the designer may not have anticipated. Users are also encouraged to consider different aspects of their activity beyond their principal use of the system. Furthermore, their responses are grouped in a systematic way that can be related via CDs to the design discourse of the system developers who receive feedback from usability studies.

Kadoda, Stone and Diaper (1999) first introduced the idea of a cognitive dimensions questionnaire. Their questionnaire only presented those CDs that they thought were relevant to the system under consideration, and to make it easier for the users to pick up the ideas, they paraphrased those CDs in terms of the system under consideration. Their results were promising, but there may be a problem here: filtering the CDs constrains the respondents to commenting on the CDs that the researcher or designer has already identified, and thereby mixes the HCI expert’s evaluation (in choosing the subset) with the users’ evaluations. This is particularly dangerous when the questionnaire designer is also the designer of the system, and will quite possibly completely overlook aspects that are very important to the users.

So we set out to develop a questionnaire which presents all the CDs, and lets the users decide which ones are relevant (Blackwell & Green 2000). We attempted to present the CDs in general terms, rather than presenting descriptions specialised to the system under consideration. On the plus side, this means (i) the users do all the work, (ii) the data only reflects their opinions, and (iii) the same questionnaire can be used for all systems. On the down side, it means that the resulting questionnaire is longer (since it has to present all the CDs, not just a subset) and possibly harder to understand, since the CDs are presented in general terms. Nevertheless, the result is now being used as an aide to the development of commercial programming tools (Clarke 2001). A canonical version of the questionnaire is available from the CDs resource site.

Cognitive Dimensions of interactive devices

We wish to avoid the impression that cognitive dimensions apply solely to purely symbolic structures such as classic notations. Although the focus of this chapter has been on notations, many interactive devices can be viewed as information artifacts where the 'notation' is the user's actions; such devices are therefore fit for analysis using CDs, offering a useful complement to other evaluative approaches.

For instance, consider a 7-day central-heating controller in which each day can be programmed independently, with say 3 heating periods per day, each defined by a switch-on time and a switch-off time[2]. The user's actions of setting the times form a notation. (An interactive mock-up of such a design is accessible via links from the CDs resource site). This design suffers from viscosity if most days – all the weekdays, say – are to be set to the same timings, the set-up process takes more effort than it ‘ought’ to. An alternative design achieves a partial reduction in viscosity by introducing an abstraction, namely a Copy function, which copies the previous day to the next day – Monday’s settings to Tuesday, Tuesday’s to Wednesday, and so on. (This version is based on a real, commercially available, device, the Horstmann ChannelPlus H17 Central Heating Controller.)

The Copy function is a transient abstraction. Although seemingly simple, experience has shown that just that one abstraction adds greatly to the difficulty of explaining how this device works -- the activity of exploratory understanding. The device also has poor visibility: the user cannot readily check the effect of the Copy function.

The above analysis is, clearly, not deep. Yet its quick, broad-brush nature is just what we would like to see available when needed. Anyone who is thinking in the terms we have described can immediately, almost trivially, recognise the features of these controllers and thereby come to a better-informed evaluation decision.

5. Case study: Evaluating a visual programming language[3]

Many diagrammatic notations represent information structures as topological graphs – system components are represented by enclosed areas (usually boxes) representing the nodes, while relationships between the components are represented by continuously connected lines extending from one box to another (wires). How usable are such box-and-wire systems? There are many variations. Apparently-small notational differences in box-and-wire languages can lead to extreme differences in the programs. Green and Petre (1996) reported a usability analysis of two commercial box-and-wire systems, Prograph and LabView[4], and how they compared to the old-fashioned Basic language.

In this section we show how one of those languages, LabVIEW, can be evaluated using the CDs framework.

Illustrating the notation

For comparative purposes, here are two versions of the same program. This program, which computes the flight path of a rocket, was derived from algorithms used by Curtis et al. for research into the comprehensibility of flowcharts; Green and Petre re-used them for their research into modern visual programming languages. First, we show the Basic version, then the LabView equivalent.

[2] A virtual version can be seen at
http://www.ndirect.co.uk/~thomas.green/workStuff/devices/controllers/HeatingA2.html

[3] This section draws extensively on Green and Petre (1995)

[4] Produced and trademarked by Pictorius Inc. and National Instruments Inc., respectively.

```
Mass = 10000
Fuel = 50
Force = 400000
Gravity = 32

WHILE Vdist >= 0
      IF Tim = 11 THEN Angle = .3941
      IF Tim > 100 THEN Force = 0 ELSE Mass = Mass - Fuel

      Vaccel = Force*COS(Angle)/Mass - Gravity
      Vveloc = Vveloc + Vaccel
      Vdist = Vdist + Vveloc

      Haccel = Force*SIN(Angle)/Mass
      Hveloc = Hveloc + Haccel
      Hdist = Hdist + Hveloc

      PRINT Tim, Vdist, Hdist
      Tim = Tim + 1

WEND
STOP
```

Figure 4 Rocket program in Basic

The LabView notation uses boxes for operations and wires for the data. Conditionals are represented as boxes lying on top of each other, only one of which can be seen at any one time. Loops are represented by a surrounding box with a thick wall.

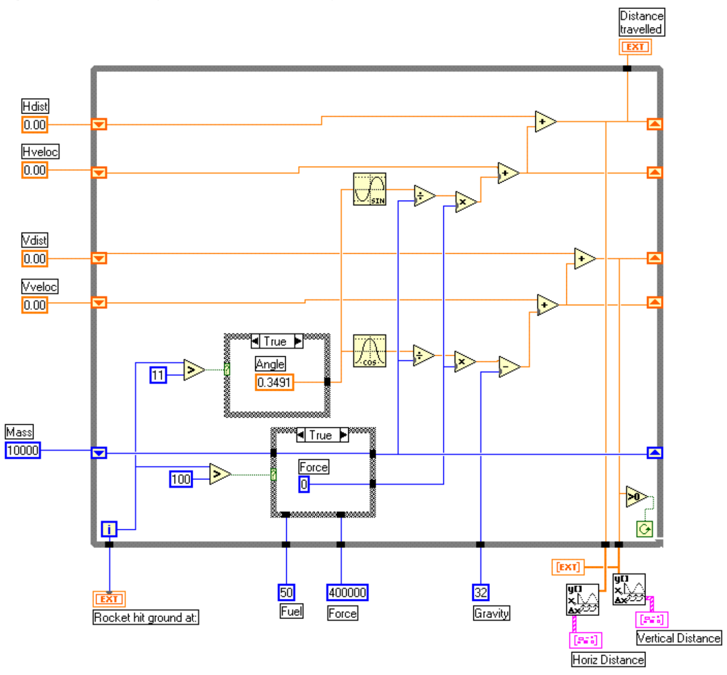

Figure 5 The rocket program in LabVIEW.

Viscosity *(definition: resistance to change)*

Green and Petre made a straw-test comparison of viscosity by modifying each of the programs in the same way, putting in a simple correction for air resistance. Because they wanted to measure the time to edit the program but not the time taken to solve the problem, the modification was worked out in advance, and they timed an experienced user modifying the original program (running in the standard environment), working from a print-out of the modification required.

Inserting the material into LabView took a surprisingly long time because all the boxes had to be jiggled about and many of the wires had to be rebuilt. Prograph was able to absorb the extra code to deal with air resistance with little difficulty — certainly less than evidenced by LabView. This was because the new code was placed in new windows, and relatively little change had to be made to existing layouts. However, this causes poor *visibility and juxtaposability* (definition: ability to view components easily). For Basic, the problem is just one of typing a few more lines. Overall time taken was as follows: LabView, 508.3 seconds; Prograph, 193.6 seconds; Basic, 63.3 seconds - an astonishing ratio of 8:1 between extremes. These are differences of a whole order of magnitude, and if the programs were larger we would expect them to increase proportionately (i.e. to increase faster for LabView, slower for Basic). Viscosity is obviously a major issue.

Hidden dependencies (defintion: important links between entities are not visible)

An important strength of the visual approach is that many of the hidden dependencies of textual languages are made explicit. Data in a textual language like Basic is transmitted via assignments and use-statements, thus:

```
x = 1
... (possibly many pages of code here...)
y = x + 3
```

Figure 6 – Hidden dependencies in BASIC

The same information in a LabView program is shown thus:

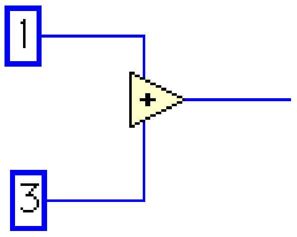

Figure 7 – Visible dependencies in LabVIEW

At least one type of hidden dependency is thereby brought into the open. The trade-off in this case, obviously enough, is the use of screen space (*diffuseness* definition: verbosity of language).

Premature commitment (definition: constraints on the order of doing things)

Green and Petre noted several types of premature commitment in the construction of these programs: commitment to layout, commitment to connections, and commitment to choice of construct.

Commitment to layout: Obviously, the visual programmer has to make the first mark somewhere on the virtual page. As the program takes shape it may become clear that the first mark was unfortunately placed, or that subsequent marks were ill-placed with respect to it and each other. Although there is always some way to adjust the layout, the viscosity may be too high for comfort. That is certainly the case with complex LabView programs.

Commitment to connections: The 2-dimensional layout of VPLs requires subroutines to be associated with their callers by ports with a definite placing. In both our target languages, one easily finds that the data terminals are not arranged in the order one wants them for a particular purpose.

Whitley and Blackwell (2001) report the astonishing fact that one of their respondents in a survey on LabView claimed to employ a junior *just to tidy up the layout of the code*:

`One respondent solved the problem by hiring extra help: "Recently, I hired a real nitpicker to clean up my diagrams while they are in progress. That saves me a lot of time and my customers get even neater diagrams than I would have done."'

That quotation tells us all we need to know about premature commitment and layout viscosity in box-and-wire diagrams.

*Commitment to choice of construct***:** When a programmer chooses a syntactic construct and then adds code to the construct, it may turn out subsequently that the construct is the wrong one (e.g. while should be changed to for). Therefore, programmers may prefer to postpone commitment to a particular construct until the code has been further developed.

Early versions of LabView suffered the problem that once a control construct had been selected, it was a tiresome chore to change it. More recent versions of LabView solve the premature commitment problem partly by postponing commitment, as well as by making it easier to change the decision (specifically, it is now possible to draw a loop around existing code, just as a for-statement can be inserted into a textual program.)

Abstraction *(definition: types and availability of abstraction mechanisms)*

The abstraction barrier (the number of new abstractions to be understood by the novice), although probably more than a simple spreadsheet system, is obviously less than say C++. Differences in this respect depend more on the choice of primitives than on the requirements of the box-and-wire notational structure itself.

LabView allows the user to choose to create new subroutines (called 'virtual instruments') but does not insist upon it: this language is therefore abstraction-tolerant. The abstraction-managing subdevice maintains a hierarchy of 'virtual instruments'; its properties will not be analysed here, but for a full treatment that step would be essential.

Layout abstractions are not in evidence; there are no easy features for group operations when re-arranging the components of a program, and much time can therefore be wasted.

Secondary Notation *(definition: extra information in means other than formal syntax)*

Designers have apparently not thought hard enough about the need to escape from the formalism. LabVIEW does not have good facilities for commenting, for example. A comment can be attached to a single item, which may seem adequate at first sight, but remember that a text language gives power to comment on a *section* of code, not just a single operation:

```
// -- compute the vertical components -- //
      Vaccel = Force*COS(Angle)/Mass – Gravity
      Vveloc = Vveloc + Vaccel
      Vdist = Vdist + Vveloc

// -- compute the horizontal components -- //
      Haccel = Force*SIN(Angle)/Mass
      Hveloc = Hveloc + Haccel
      Hdist = Hdist + Hveloc
```

Figure 8 – Secondary notation in the form of comments

Looking back to the LabView program, how would those comments be adapted?

Nor can LabView achieve the same degree of control over spatial layout that Basic achieves over textual layout. The Basic program has been organised to bring related operations into close proximity (which is what makes those comments so effective), but in the visual languages the severity of the constraints on laying out the boxes to avoid clutter and crossings makes it impossible to achieve the same results.

Visibility and Juxtaposability *(definition: ability to view components easily)*

The visibility of data flow in the LabView language is excellent.

Where LabView meets trouble, however, is in the juxtaposition of related control branches. The two branches of a conditional cannot both be viewed at once, and although a single mouse click will take you to the next branch, the effect on comprehensibility is very worrying.

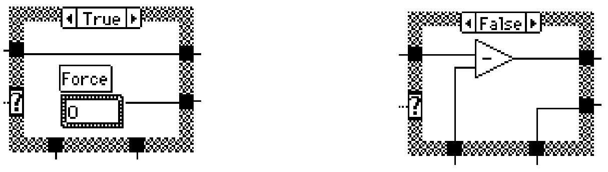

Figure 9 A LabView conditional from the rocket program, showing both arms. In the LabView environment, only one arm is visible on screen at any one time.

Conclusions

The most striking features of these comparisons are on the one hand, the extraordinarily high viscosity of box-and-wire notations and the very poor facilities for secondary notation, and on the other hand, the remarkable freedom from hidden dependencies. These give box-and-wire programs a very different feel from textual languages, and indicate some targets for would-be improvers to aim for.

6. Current status

The CDs framework has been adopted by a broad community of researchers, and also by a few practitioners. The bibliography of the CDs resource site currently lists about 60 publications on cognitive dimensions. Future progress on the framework is likely to focus on four issues: making the benefits available to a broader range of software development professionals, increasing the theoretical rigour in the foundations of the framework, filling in gaps in coverage, and providing new analysis tools based on CDs. We describe some of these continuing research efforts in the following sections.

Dissemination

Most dissemination of the CDs framework continues to be centred on academic venues, via invited talks and tutorials at related conferences. A few specialised symposia have been held, devoted to discussion of ongoing research on the CDs framework. There is no centralised organisation for these meetings, which have been hosted at institutions with a concentration of CD researchers such as the University of Hertfordshire. Several universities are teaching final year undergraduate courses or masters-level courses including CDs as an analysis and design technique. Most of this work is coordinated via the CDs resource site.

Clarification and formalisation

The dimensions are at present defined in informal terms, and in consequence there are areas of overlap and uncertainty. It is imperative to clarify their descriptions and if possible to reduce their number.

Roast and Siddiqi (2000) have developed a system-modelling approach, using system abstractions to lead to closer definitions of the precise meaning of various cognitive dimensions. Detailed consideration of their careful work would take up too much space here, unfortunately, but some flavour of it can be given by looking at the route taken by Roast in examining premature commitment.

Roast's approach provides interpretations based on

- the goals and sub-goals which users may achieve,
- the user inputs which enable users to satisfy different goals,
- the objects central to characterising the goals to be achieved.

Roast describes premature commitment as “the user having to satisfy the secondary goal prior to achieving the primary goal.” To develop a formal estimation of premature commitment he focuses on three facets of the concept: the characterisation of how the secondary goal is unavoidable; the relationship between the secondary and primary goals which indicates the inappropriateness of having to commit to the secondary goal first, and the initial conditions. This leads to a new distinction between weak and strong premature commitment, and thence to a detailed formal analysis of the relationship between premature commitment and viscosity. Finally, in a small case study, the formal analysis is applied to an editor used for the specification language Z, and is shown to conform to the difficulties experienced by certain users.

Blackwell is continuing work on a formalised description of user activities, based on an economic model of attention investment (Blackwell & Green 1999). This model describes the cognitive process of attention to a task in terms of the potential return on investment when that work results in later time savings. All programming work can be described as a short term investment of attention to writing the program, in return for a longer-term saving of time through automated tasks. The CDs of the notation and environment being used directly affect the factors controlling return on investment in this analysis.

Coverage

Green and Petre (1996) under the heading of “Future progress in cognitive dimensions”, observed that the framework was incomplete – but not in the sense that more dimensions were urgently needed. Rather it emphasised the need for formalisation and applicability. Nevertheless, new dimensions do get proposed from time to time. Some of these proposals have been published, but more have been discussed informally. A recent paper outlined future approaches to the process of identifying and defining new Cognitive Dimensions, illustrating this discussion with descriptions of several candidates (Blackwell, Britton et. al. 2001). Other aspects of the framework, especially the enumeration of design maneuvers and tradeoffs, but also fundamentals such as the description of user activities, continue to be refined and expanded.

Analysis tools

Green (1991), much developed by Green and Benyon (1996), offered an extended entity-relationship model of information artefacts in which certain of the cognitive dimensions could fairly easily be identified and in some cases metricated. Viscosity lends itself quite well to this approach. An ERMIA (entity-relationship modelling for information artefacts) model of conventional music notation looks like this:

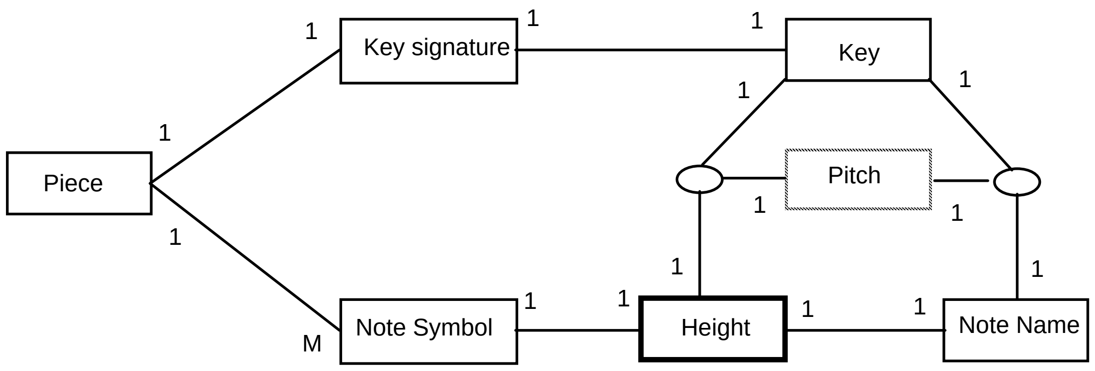

Figure 10 ERMIA Model of music notation

indicating that the piece of music contains many note symbols, each of which has a height on the staff (the black border indicates a perceptual cue) and a note name. To transpose the key of a piece, every note must be rewritten. The approach even offers a symbolic computation of the cost of change:

```
Transpose Staff Notation
      write Key'                          change key signature
      foreach Note:                       change all notes
            read Note
            compute Note'
            write Note'

Total actions:   #(write actions) = #(Notes)
Working memory storage:
      1 place-holder , to keep track of notes in the original document
```

For this structure, V = O(n), where n is the number of note symbols. (The value of n can run into the thousands.) Note that an alternative representation for music, known as 'tonic sol-fa', reduces the viscosity down to a single action!

A very different approach was taken by Yang et al. (1997) who defined simple, a priori yardsticks by which to quantify some of the dimensions in the special case of visual programming languages. They realised that for practical use benchmarks with real numbers attached would be more useful for some purposes than mere discussion tools, so they set out to define metrics for some of the dimensions. They sensibly restricted their aims to those dimensions that they thought would be of most interest for their purposes of designing and improving visual programming languages, and their first decision was to concentrate on the static part of a program representation rather than its dynamics, arguing that obtaining a usable static representation was an essential first step.

Eighteen benchmarks are defined in their paper. Examples are:

Visibility or hiddenness of dependencies:

D1: (sources of dependencies explicitly depicted) / (sources of dependencies in system)

D2: The worst-case number of steps required to navigate to the display of dependency information

Visibility of program structure:

PS1: Does the representation explicitly show how the parts of the program logically fit together? (Yes/No)

These benchmarks, though admittedly somewhat on the crude side, were applied by the authors to different languages and helped them find overlooked usability problems and encouraged them to redesign the languages. They encourage others to find similar domain-specific benchmarks for their own purposes.

Beyond CDs: Misfit analysis

Many of the cognitive dimensions may be seen as examples of misfits between the user's view of the domain (related to a mental model) and the 'device's' view. Take viscosity as an instance: the user thinks of a single direct operation but the device requires an indirect operation, or a large number of operations. Other misfits are not so readily expressible as cognitive dimensions, for instance inexpressiveness: the user wishes to do something that is part of the conceptual domain but cannot be expressed using the device. *Example:* electronic organisers cannot indicate relative importance of engagements, nor links between them, which are important components of diary use (Blandford & Green 2001). Thus, the concept of a misfit may be more general than the concept of a cognitive dimension.

Ontological Sketch Modelling (Blandford & Green 1997) is an attempt to develop the idea of 'misfit' analysis. It provides a simple and deliberately sketchy formalism in which to express the user's conceptual models of the device, the domain and working practices (that is: how the device fits in with the way the user works). Analysing the degree of fit between these can reveal potential problems of a semantic type that are not revealed by existing HCI approaches.

Misfits cannot be revealed by any approach to HCI that focuses solely on either the user or the device. Traditional task-centred user-modelling for HCI has some very effective results, but it cannot reveal misfits because it does not explicitly consider how the user's conceptual model of the domain and device relates to the model imposed by the device.

In OSM the modeller describes the visible entities, their attributes, and how they are linked within the device; and also describes the entities contained in the user's conceptual model and those embodied within the device that the user needs to be aware of if they are to use the device effectively. The resulting entities may be private to the device (the user cannot alter them), or they may be private to the user (the device does not represent them), or they may be shared (accessible to both the device and the user). All communication between the two worlds of user and device takes place through the shared entities. If the user-private entities do not fit well onto the shared entities, the device will have usability problems. Similarly, if the device-private entities are difficult to discover or understand, the user is likely to have difficulty learning to work with them. However, the cognitive dimensions were presented entirely informally, and the method of analysis was no more than asking a designer to think carefully about each dimension in turn. With OSM, some of the dimensions can be evaluated algorithmically, once a description of the artefact has been constructed.

7. Further reading

Most research and tutorial material related to CDs is available from the CD resource site:

http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/

The site includes an online bibliography of relevant research publications, a variety of introductory papers at different levels, and links to ongoing research. Some key starting points include the CDs tutorial by Green and Blackwell (1998), an extended research presentation by Green and Petre that was published in the Journal of Visual Languages and Computing (Green & Petre 1996), and a description of the use of user questionnaires for CD evaluation (Blackwell & Green 2000). The bibliography is annotated to direct the reader to other specialist topics.

Related research on usability and cognitive aspects of notational systems continues to be published in venues such as the IEEE conference on Human-Centred Computing, the Journal of Visual Languages and Computing, the ACM conference on computer-human interaction (CHI) and conferences devoted to cognitive science, psychology of programming, information design and diagrammatic reasoning. Most of these topics also have mailing lists, online bibliographies, and resource sites devoted to them:

http://www.hcrc.ed.ac.uk/gal/Diagrams/ - Diagrams resource site

http://www.ppig.org/ - Psychology of Programming resource site

http://hcibib.org/ - HCI bibliography

References

Apple Computer, Inc. (1992). *Macintosh Human Interface Guidelines*. Reading, MA: Addison-Wesley.

Blackwell, A.F. (1996). Metacognitive Theories of Visual Programming: What do we think we are doing? In Proceedings IEEE Symposium on Visual Languages, pp. 240-246.

Blackwell, A.F. (2000). Dealing with new Cognitive Dimensions. Paper presented at Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community. University of Hertfordshire, 8 December 2000.

Blackwell, A.F. & Green, T.R.G. (1999). Investment of attention as an analytic approach to Cognitive Dimensions. In T. Green, R. Abdullah & P. Brna (Eds.) Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11), pp. 24-35.

Blackwell, A.F. & Green, T.R.G. (2000). A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell & E. Bilotta (Eds.) Proceedings of the Twelth Annual Meeting of the Psychology of Programming Interest Group, 137-152.

Blackwell, A.F., Britton, C., Cox, A., Green, T.R.G., Gurr, C., Kadoda, G., Kutar, M.S., Loomes, M., Nehaniv, C.L., Petre, M., Roast, C., Roe, C., Wong, A. & Young, R.M. (2001). In M. Beynon, C.L. Nehaniv & K. Dautenhahn (Eds.) Cognitive technology: Instruments of mind. Springer Verlag, pp. 325-341.

Blandford, A.E. & Green, T.R.G. (1997) OSM: an ontology-based approach to usability evaluation. Proceedings of Workshop on Representations, Queen Mary & Westfield College (July 1997).

Blandford, A.E. & Green, T.R.G. (2001). From tasks to conceptual structures: misfit analysis. In Proc. IHM-HCI2001.

Booch, G., Rumbaugh J. & Jacobson I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Burnett, M., Cao, N., and Atwood, J. (2000) Time in grid-oriented VPLs: just another dimension? IEEE International Symposium on Visual Languages (VL2000), Los Alamitos, CA: IEEE Computer Society, pp 137-144

Cheng, P.C. (1998). AVOW diagrams: A novel representational system for understanding electricity. In *Proceedings Thinking with Diagrams 98: Is there a science of diagrams?* pp. 86-93.

Clarke, S. (2001). Evaluating a new programming language. In G. Kadoda (Ed.) Proceedings of the 13th Annual Meeting of the Psychology of Programming Interest Group.

Eckert, C.M. and Stacy, M.K. (2000) Sources of inspiration: a language of design. *Design Studies*, 21(5).

Fish, J. & Scrivener, S. (1990). Amplifying the mind's eye: Sketching and visual cognition*. Leonardo*, **23**(1), 117-126.

Fitter, M. & Green, T.R.G. (1979). When do diagrams make good computer languages? International Journal of Man-Machine Studies, 11(2), 235-261.

Goldschmidt, G. (1991). The dialectics of sketching*. Creativity Research Journal*, **4**(2), 123-143.

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

Green, T.R.G. (1991) Describing information artifacts with cognitive dimensions and structure maps. In D. Diaper and N. V. Hammond (Eds.) Proceedings of HCI'91: 'Usability Now', Annual Conference of BCS Human-Computer Interaction Group. Cambridge University Press.

Green, T. R. G. (1999) 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.

Green, T.R.G. & Blackwell, A.F. (1998). Design for usability using Cognitive Dimensions. Tutorial session at British Computer Society conference on Human Computer Interaction HCI'98. Current revisions are maintained online at: http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf

Green, T.R.G., Bellamy, R.K.E. & Parker, J.M. (1987). Parsing and gnisrap: A model of device use. In G.M. Olson, S. Sheppard & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop. Norwood, NJ: Ablex, pp. 132-146.

Green, T.R.G. & Benyon, D. (1996.) The skull beneath the skin: entity-relationship models of information artifacts. Int. J. Human-Computer Studies, 44(6) 801-828.

Green, T.R.G. & Petre, M. (1992). When visual programs are harder to read than textual programs. In G.C. van der Veer & S. Bagnarola (Eds.), Proceedings of ECCE-6 (European Conference on Cognitive Ergonomics).

Green, T.R.G. & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' approach. Journal of Visual Languages and Computing, 7,131-174.

Green, T.R.G., Petre, M. & Bellamy, R.K.E. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In J. Koenemann-Belliveau, T.G. Moher & S.P. Robertson (Eds.): Empirical Studies of Programmers: Fourth Workshop Norwood, NJ: Ablex, pp. 121-146.

Kadoda, G., Stone, R. & Diaper, D. (1999). Desirable features of educational theorem provers – a cognitive dimensions viewpoint. In T.R. G. Green, R. Abdullah & P. Brna (Eds.) Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group (PPIG-11), pp. 18-23.

Kress, G. & van Leeuwen, T. (1996). Reading images: The grammar of visual design. London: Routledge

Myers, B.A. (1986). Visual Programming, Programming by Example, and Program Visualization: a taxonomy, Proc. CHI 86, pp.59-66.

Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces, *Proceedings of ACM CHI'90* Conf. (Seattle, WA, 1-5 April), 249-256.

Pressman, R. S. (1997). Software Engineering: A Practitioner's Approach. McGraw-Hill.

Roast, C. R. and Siddiqi, J. I. (2000) Formal comparisons of program modification. In VL-2000, Proc. of 2000 IEEE International Symposium on Visual Languages, Los Alamitos, CA: IEEE Computer Society, 2000.

Robinson, A. (1995). The story of writing. London: Thames and Hudson.

Shimojima, A. (1996). Operational constraints in diagrammatic reasoning. In G. Allwein & J. Barwise (Eds), Logical reasoning with diagrams. Oxford University Press, pp. 27-48.

Suwa, M. & Tversky, B. (1997). What do architects and students perceive in their design sketches? A protocol analysis. *Design Studies*, **18**, 385-403.

Wharton, C., Rieman, J., Lewis, C., & Polson, P. 1994 The cognitive walkthrough method: a practitioner's guide. In J. Nielsen and R. Mack (Eds.), *Usability inspection methods*. John Wiley & Sons, Inc., New York, NY, 1994.

Whitley, K.N. and Blackwell, A.F. (2001). Visual programming in the wild: A survey of LabVIEW programmers. Journal of Visual Languages and Computing, 12(4), 435-472.

Winograd, T., Bennett, J., De Young, L. and Hartfield, B. (eds.), (1996). Bringing design to software, Addison Wesley.

Yang, S., Burnett, M. M., DeKoven, E. and Zloof, M. (1997) Representation design benchmarks: a design-time aid for VPL navigable static representations. Journal of Visual Languages and Computing, 8 (5/6), 563-599.