

Cognitive Dimensions of Information Artefacts: a tutorial

Thomas Green and Alan Blackwell

Version 1.2 October 1998

This document can be downloaded via links at
<http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/>

Copyright © T. R. G. Green and A. F. Blackwell, 1998

Preface to the tutorial

This tutorial, originally prepared for the BCS HCI Conference of 1998, will provide a concise introduction to Green's "Cognitive Dimensions", with illustrative practical applications.

The Cognitive Dimensions Framework is one of the few theoretical approaches that provides a practical usability tool for everyday analysts and designers. It is unique in stressing simple, cognitively-relevant system properties such as *viscosity* and *premature user commitment*, and it illustrates important trade-offs and design manoeuvres.

The tutorial is intended for HCI practitioners and consultants, especially those actively involved in design of novel user interfaces, and for HCI researchers with an interest in system evaluation and implementation. It provides:

- a brief comparison of types of user activity, and how each is best supported by what profile of dimensions;
- many snapshot examples to illustrate the dimensions;
- example analyses of commercially-available systems;
- a discussion of standard 'design manoeuvres' and their associated trade-offs;
- a set of interactive, web-available miniature examples to illustrate design alternatives and their effects;
- a Further Reading section for those wishing to extend their knowledge.

Since this tutorial is practitioner-oriented we shall focus on the more immediately practical parts of Green's framework. On completion, you will have a vocabulary in which to analyse systems, and a checklist with which to consider how a system relates to its intended type of user activity; you will be able to apply the ideas to both interactive systems and non-interactive (paper-based) information representations; you will be able to choose design manoeuvres to ameliorate usability problems, and you will be aware of the trade-offs that each manoeuvre entails.

Part 1 describes the framework. Part 2 applies it to some commercially-available examples, to show what types of insight can be expected from it. Part 3 gives short descriptions of some interactive toy examples, designed to illustrate differences between design decisions and how one cognitive dimension can be traded against another; these examples are available over the web.

About the authors

Thomas Green has published extensively on cognitive and empirical approaches to HCI and has co-authored several contemporary techniques (TAG, ERMIA, CDs, and watch out for Ontological Sketch Models!). Ex-APU Cambridge, now Honorary Research Fellow at the University of Leeds. Address: Computer-Based Learning Unit, University of Leeds, Leeds LS2 9JT, U.K.

thomas.green@ndirect.co.uk
<http://www.ndirect.co.uk/~thomas.green>

Alan Blackwell has a wide background in AI, cognitive science, and professional software engineering. He was organiser of the first "Thinking with Diagrams" workshop and is currently researching on visual programming languages. Address: MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge CB2 2EF, U. K.

alan.blackwell@mrc-apu.cam.ac.uk
<http://www.mrc-apu.cam.ac.uk/~alan.blackwell>

CONTENTS

Preface to the tutorial	2
PART 1: THE FRAMEWORK	4
Introduction	5
Cognitive Dimensions: the full set	11
Viscosity	12
Hidden Dependencies	17
Premature Commitment and Enforced Lookahead	21
Abstractions, Abstraction Hunger, and the Abstraction Barrier	24
Secondary Notation	29
Visibility & Juxtaposability	34
The Remaining Dimensions	39
Cognitive Relevance, Trade-offs, and Design Manoeuvres	42
Further developments in cognitive dimensions	44
PART 2: REAL SYSTEMS	46
A Diary System	47
Two Visual Programming Languages	52
References (Pts 1 and 2)	61
PART 3: INTERACTIVE PRACTICE EXAMPLES	62
The Widgets	64
Form-Filling and Menu Choices	65
Controlling the Heating Controller	66
Number Games with Telephones	67
Tiling in Style	69
FURTHER READING	71
Index of examples	75

Part 1: The Framework

In this part we describe the cognitive dimensions approach – its purpose, its framework, the individual components (the dimensions themselves), and its relevance to design choices. Following parts address applications to real systems (part 2) and interactive examples for class discussion and analysis (part 3).

Introduction

Background and Purpose

What are ‘Cognitive Dimensions’ for?

Cognitive dimensions are a tool to aid non-HCI specialists in evaluating usability of information-based artefacts (summative evaluation). Since they are addressed to non-specialists, they consciously aim for broad-brush treatment rather than lengthy, detailed analysis. Their checklist approach helps to ensure that serious problems are not overlooked.

They can also be used by designers to prompt possible improvements (formative evaluation). Many design choices, from widely different domains, can be brought together and are then seen to be standard ‘design manoeuvres’ intended to ameliorate particular usability problems. Each manoeuvre that improves one aspect will be associated with a likely trade-off cost on other aspects, and the dimensional approach helps to elucidate the trade-offs.

Historically, the framework was introduced by Green (1989, 1991), who emphasised the generality of this approach, applicable over all domains where information representation was used. A lengthy analysis of one particular domain, that of visual programming languages, by Green and Petre (1996), is the most detailed presentation of the framework currently published. For more details, see the section entitled ‘Further Reading’.

What do they work on?

The cognitive dimensions approach applies to both interactive and static systems – to anything, as long as it’s an ‘information artefact’. Information artefacts are the tools we use to store, manipulate, and display information. They comprise two classes:

- *interactive* artefacts, such as word-processors, graphics packages, mobile telephones, radios, telephones, central heating control systems, software environments, VCRs,
- *non-interactive* artefacts, such as tables, graphs, music notation, programming languages, etc.

In this respect the approach is unlike most evaluation methods in HCI, which are solely aimed at interactive devices, such as editors and the like. The focus of those approaches is on the *process of interaction*. Therefore, they do not work well on non-interactive artefacts.

Discussion tools, not deep analyses

Cognitive dimensions are ‘discussion tools’. They give names to concepts that their users may only have half-formulated, or may not have formulated at all.

Giving names to concepts (lexicalisation) is essential to serious thought and discussion. Once names are given, discussants do not have to explain each idea every time they use it; they can refer to the idea knowing that the other party will understand it. Paradigmatic examples help to make the concepts concrete. Comparisons of trade-off costs become easier. Above all, a checklist can be constructed, to make sure that an evaluative discussion has not overlooked important aspects.

Because these dimensions name concepts that are familiar, even if not well-formulated by most users, the approach is easy to grasp. Learning to think fluently in these terms may take practice, but the same is true of any conceptual system.

Cognitive dimensions will not give their users powerful mathematical tools for deep analysis. Such analytical tools have plentiful uses, but they will never serve for broad-brush, non-specialist use. And because the cognitive dimensions approach covers a wide range of very different dimensions, an equally wide variety of mathematical analyses would be necessary to achieve the same coverage by purely analytical techniques; there would need to be techniques drawn from parsing theory, from

analysis of information structures, from human reliability modelling, perceptual theory, and so on. The result would be horrendous, a system that was too hard to learn, too slow to use, and too opaque to talk about to non-specialists.

How do Cognitive Dimensions relate to more traditional approaches?

The cognitive dimensions analysis is a very different philosophy from such approaches as GOMS and the Keystroke Level Model (Card et al., 1981) and Heuristic Evaluation (Nielsen and Molich, 1990)..

The Keystroke Level Model delivers predictions of overall time taken for a specified task. A detailed task analysis is needed, usually based on GOMS, a considerable investment in time and expertise. The value of task analysis has been much debated; users often display inventive unexpected approaches that were not conceived by the task analysts.

Heuristic Evaluation identifies poor screen design, badly-phrased wording, and the like. Much quicker than task analysis, it needs the services of up to three HCI experts viewing the displays of the fully-implemented artefact or of a reasonable mock-up. There is no theoretical foundation, but within its scope it produces comprehensible and helpful suggestions. The approach has no way to consider effects of the underlying information structure, which is a serious shortcoming.

The Cognitive Dimensions approach should be seen as complementary to other approaches. It is easy to learn and quick to use; it identifies ‘sub-devices’ and activity types, unlike other approaches; and it can be applied at *any stage of design*, from idea to finished artefact. Because the ideas are so simple, they make sense to ordinary users as well as to specialists.

<i>Cognitive Dimensions</i>	<i>traditional approaches</i>
broad-brush	highly detailed
quick to learn	specialist training needed
quick to apply	lengthy analysis
applicable at any stage of design	requires full task analysis (GOMS/KLM) or fully-implemented design or mock-up (Heuristic Eval)
differentiates user activity types	all activity evaluated identically
multi-dimensional	single dimension
vague	precise metric
comprehensible to non-specialists	only the metric is comprehensible - not the basis for it

Figure 1 Comparison of Cognitive dimensions and traditional evaluative approaches.

What does the approach deliver?

The CDs approach avoids any kind of simplified ‘bug hunting’ or ‘overall difficulty measure’. Instead, the information artefact is evaluated along 13 different dimensions, each of which is cognitively relevant, giving a ‘profile’. The profile determines the suitability for various tasks.

The dimensions are not intrinsically good or bad. Different types of user activity are best supported by different profiles.

Four types of activity

Designing a garden is not like designing a bridge. Garden design is very aesthetic, and part of the job is exploring the options, changing one’s mind, feeling one’s way to a new vision. Bridge design is less inspirational and very much more safety-critical. These are different types of user activity, and they are likely to be supported by different kinds of software or drafting tools.

It would be nonsense to claim that all information artefacts should satisfy some single set of criteria. We shall distinguish four main classes of activity, and as we explain the framework we shall develop a preferred profile for each type of activity. The four types we distinguish are:

incrementation:	adding a new card to a cardfile; adding a formula to a spreadsheet
transcription:	copying book details to an index card; converting a formula into spreadsheet terms
modification:	changing the index terms in a library catalogue; changing the layout of a spreadsheet; modifying the spreadsheet to compute a different problem
exploratory design	typographic design; sketching; programming on the fly ('hacking')

Figure 2 The four types of activity distinguished in this framework

These activity types are the pure and undisturbed cases. Sometimes the situation imposes extra demands – for example, the forms-based device illustrated in Figure 18 (page 37) is intended for incrementation, but the poor design forces users to hut through their previous entries in a way that approaches the problems of exploratory design.

Each activity is best supported by its own profile of CDs. The profiles will be gathered together as the discussion proceeds. (If impatient, look ahead to Figure 19 on page 42.)

Why 'cognitive dimensions'?

The approach deserves to be called cognitive because it focuses on usability aspects that make learning and doing hard for mental (not physical) reasons. Button size, for instance, is a physical, not a cognitive, issue although it has been included in certain versions of 'cognitive user modelling'. On the other hand, the degree to which the system makes users translate one operation in the head into a multitude of individual actions is assuredly cognitive, since it relates to the user's conception of an operation (and ironically, this factor is not included in most versions of 'cognitive user modelling').

Aesthetic or emotive aspects of usability, important as they are, have no place in this framework. Nor does the analysis of context-of-use, an aspect of usability whose importance is increasingly becoming recognised. These omissions are not implicit assertions that they have no relevance. Our belief is quite the contrary, but this particular framework does not lend itself readily to being extended in those directions, which we believe need to be analysed in other ways.

More problematic to some is why we believe the term 'dimension' to be appropriate. Comparison with the dimensions of physics is knowingly invited by this choice, justified, in our opinion, by some important similarities:

- the cognitive dimensions are conceptual idealisations, just like velocity or potential energy
- for our purposes, entities are fully described by their position on the various dimensional scales
- the dimensions are conceptually independent, just as mass is conceptually independent of length and time
- for real entities, the dimensions are only 'pairwise' independent, meaning that although any given pair can be treated as independent, a change on one dimension usually entails a change on some other dimension, in a way that we shall describe shortly.

An important corollary follows from the last point, namely that changing the structure of an artefact is a matter of trading-off one disadvantage against another: just like physics, cussedness is constant.

Trade-offs and Pairwise Independence

Take a constant mass of gas, with 3 'dimensions': temperature, pressure and volume.

If you heat the gas, the temperature rises, and it tries to expand. If the volume is held constant, it can't expand, so the pressure increases. If the pressure is to be kept constant, the volume must increase.

So although pressure, temperature, and volume are conceptually independent, for physical objects they are only pairwise independent. Within reason, any combination of pressure and temperature may be obtained, but between them they determine the volume; and likewise for all other combinations.

Many of the cognitive dimensions are similarly pairwise independent: any two can be varied independently, as long as some third dimension is allowed to change freely.

The structure of information artefacts

Before explaining the dimensions themselves we need to explain the model of information artefacts and their construction and manipulation.

To present the framework it is necessary to introduce the concepts of ‘notation’, ‘environment’ and ‘medium’. To motivate these concepts we shall briefly present one example of a dimension, the most convenient being ‘viscosity’.

A viscous system is one that is hard to modify. (This is over-simplistic – at least two types of viscosity can be distinguished – but it will serve for the illustration.)

- Text written with a word-processor is easy to modify; text written with a pen and paper is much harder to modify. These are different *environments*.
- Inserting a new paragraph into a novel is much easier than inserting a new page into a very formal document with numbered paragraphs and manifold cross-references. These are different information structures, or *notations*.
- Changing a paragraph or even a word is much easier on paper, where the whole text is visible and any part of it can quickly be reached, than when it is being dictated to a scribe, when only the most recent word can easily be changed (“sorry, I’ll say that again”). These are different *media*.

The notation comprises the perceived marks or symbols that are combined to build an information structure. The environment contains the operations or tools for manipulating those marks. The notation is imposed upon a medium, which may be persistent, like paper, or evanescent, like sound.

The cognitive dimensions assess the process of building or modifying information structures. Since that process is affected by all three of the notation, the environment, and the medium, it is necessary to take them all into account. A notation cannot be said to possess viscosity; only a system, comprising all three, can be viscous. (In what follows we shall assume that the medium is persistent, unless specifically stated otherwise.)

The interpretation of this framework is immediately apparent if we restrict ourselves to non-interactive artefacts, such as a programming language: the notation is the language itself, the environment is the editing system, and the medium is normally the computer screen (but occasionally we may have to give programming support by telephone, and then the differences between media become very marked).

Interaction languages as a form of notation

It may not be so easy at first to see how to apply the framework to interactive artefacts, such as using a telephone. Consider first the ‘information structure’ that is to be built; we can take it to be the number that is to be dialled. The notation is the set of digits, possibly including the star and hash keys. The environment may simply allow each number on the keypad to be pressed, or it may include a ‘redial last number’ feature; there is also an action (replacing the handset) that cancels everything done so far. The medium is a persistent one, in that the telephone number is being constructed inside the memory of the telephone system.

Thus, we model all information artefacts in terms of notations and their use. Non-interactive artefacts, such as timetables, are directly expressed in the notation. Interactive artefacts are controlled by an ‘interaction language’ whose commands are the notation.

Layers

Now consider a slightly more complicated example, building a program.

There are two layers. In one layer, the notation is the programming language, the environment is the editor, and the medium is the computer memory, which is persistent, since the program is built up there.

The other layer is the editing layer. Here, the notation is keystrokes and mouse actions, the environment is the keypad, and the medium is transient – the notation is translated into editor actions, such as selecting a word or deleting a character in the program.

Each layer has its own dimensions. Viscosity of the program text is determined by the programming language and by whether the editor has tools specifically designed for that language. Viscosity of the editor layer is determined by how a user intention, such as selecting a given word, is expressed as a sequence of actions, and whether it is easy to recover from a slip in the execution of those actions.

A serious analysis of layers is beyond the scope of what can be covered in this tutorial, but it is necessary at times to be aware of the different layers involved or to risk analysing the wrong level of interaction.

Sub-devices

The last component of the framework is the notion of a *sub-device*. A sub-device is a part of the main system that can be treated separately because it has different notation, environment, and medium.

The simplest sub-device might be a pad of paper, considered as part of an information system whose main components are software. Its function might be to record notes about the main information structure, such as list of identifiers not easily obtained from the main environment; or its function might be to allow work to be sketched out before being transcribed to the main system – not infrequently, pencil and paper is a better system for exploratory design than the main system for which the design is intended.

One particular type of sub-device must be mentioned here, even though it will not receive full treatment until considerably later: we call it the ‘abstraction manager’. Any system that allows users to create new terms, such as macros, needs some kind of manager to record the names and definitions of the macros and perhaps to allow them to be edited. Even a humble telephone with an internal memory of ten numbers needs a simple manager so that the user can associate a quick-dialling code with some oft-used telephone number. The sub-device in these cases builds persistent structures out of the notation of actions (the commands that the macro executes or the digits of the keypad), but it does not execute the actions – that is the job of the host device.

Summary of the framework

The Cognitive Dimensions framework gives a broad-brush usability profile, intended to provide discussion tools rather than detailed metrics. The 13 or so dimensions can be applied to any type of information device, whether interactive or not, allowing similarities to be detected in very different types of device.

All forms of interaction are seen as building or modifying an information structure. Non-interactive (e.g. paper-based) information, such as the structure of a timetable, is assessed in terms of the ease of creating or changing it. The operation of interactive devices is seen as creating commands in an 'interaction language'; these commands are the information structures that are to be built or modified.

Usability depends on the structure of the notation and the tools provided by the environment. Thus, the dimensions apply to the system as a whole, taking into account the tools provided as well as the notation or the information structure. They also depend on what 'persists': in a conventional program, the names of actions persist and can be edited, but in an interactive device the effects of actions persist rather than their names.

Some forms of evaluation are geared towards 'bug-hunting' or towards predicting time to perform a task. The cognitive dimensions approach is not so simplistic. The dimensions are not intrinsically 'good' or 'bad'; together they define a hyperspace, and for particular types of user activity, different regions of that hyperspace are appropriate.

Four types of user activity are distinguished in this document:

- incrementation – adding further information without altering the structure in any way (e.g. entering a new file card);
- modification – changing an existing structure, possibly without adding new content;
- transcription – copying content from one structure to another structure;
- exploratory design – combining incrementation and modification, with the further characteristic that the desired end state is not known in advance.

Each type of user activity will be best served by a different set of values on the dimensions.

The dimensions are supposedly independent, in the sense that for a given system, it is possible to redesign it to change its position on one given dimension A without affecting its position on another dimension B. Independence is only pairwise, however – almost certainly some other dimension C will be affected (and vice versa, A can be changed without changing C, but something else, e.g. B, will have to change). Redesign is therefore an exercise in choosing trade-offs and making compromises (see below).

Cognitive Dimensions: the full set

This list of cognitive dimensions has been arrived at by exploring a variety of devices and looking for important differences. There is no guarantee that it is complete or fixed.

In this document we shall only explore a subset, namely those that are less psychological in character. Others, such as error-proneness and role-expressiveness, demand more understanding of cognitive psychology than it would be convenient to present here.

Dimensions treated in this tutorial

Abstraction	types and availability of abstraction mechanisms
Hidden dependencies	important links between entities are not visible
Premature commitment	constraints on the order of doing things
Secondary notation	extra information in means other than formal syntax
Viscosity	resistance to change
Visibility	ability to view components easily

Dimensions not treated in this tutorial

Closeness of mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard mental operations	high demand on cognitive resources
Progressive evaluation	work-to-date can be checked at any time
Provisionality	degree of commitment to actions or marks
Role-expressiveness	the purpose of a component is readily inferred

Viscosity

Definition

Resistance to change: the cost of making small changes.

Repetition viscosity: a single goal-related operation on the information structure (one change 'in the head') requires an undue number of individual actions,.

Knock-on viscosity: one change 'in the head' entails further actions to restore consistency

Thumbnail illustrations

Repetition viscosity: Manually changing US spelling to UK spelling throughout a long document

Knock-On viscosity: inserting a new figure into a document creates a need to update all later figure numbers, plus their cross-references within the text; also the list of figures and the index

Explanation

Viscosity in hydrodynamics means resistance to local change – deforming a viscous substance, e.g. syrup, is harder than deforming a fluid one, e.g. water. It means the same in this context. As a working definition, a viscous system is one where a single goal-related operation on the information structure requires an undue number of individual actions, possibly not goal-related. The ‘goal-related operation’ is, of course, at the level that is appropriate for the notation we are considering. Viscosity becomes a problem in opportunistic planning when the user/planner changes the plan; so it is changes at the plan level that we must consider. This is important, since changes at a lower level can sometimes be pure detail (inserting a comma, for instance), giving a misleading impression of low viscosity. So, in general, *viscosity is a function of the work required to add, to remove, or to replace a plan-level component of an information structure.*

Note that viscosity is most definitely a property of the system as a whole. Certain notations give the potential for viscosity (e.g. figure numbers) but the environment may contain tools that alleviate it by allowing aggregate operations (e.g. a smart word-processor with auto-number features). See below for associated trade-offs.

Also note that viscosity may be very different for different operations, making it hard to give a meaningful overall figure.

Cognitive relevance

Breaks train of thought, gets too costly to change or to take risks; encourages reflection and planning, reduces potential for disastrous slips and mistakes.

Acceptable for transcription activity and for simple incrementation activity, but problematic for exploratory design and for modification activity.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>viscosity</i>	<i>acceptable</i> <i>(because one shouldn't need to make changes)</i>	<i>acceptable</i>	<i>harmful</i>	<i>harmful</i>

Cost Implications

It is easy for designers to view a device as a pure incrementation system and overlook the need for restructuring. *This is a serious and far too frequent mistake.* So a computer-based filing system may readily allow new categories to be added, but have very little provision for redefining categories and re-categorising the objects within them.

The worst problems come when viscosity is combined with premature commitment. The user has to make an early guess (that's the premature commitment) but then finds that correcting that guess is time-consuming or costly (that's the viscosity).

Types and Examples

Repetition viscosity is a frequent problem in structures which exist in the user's mind rather than being recognised by the system. A **collection of files** making up one document in the user's mind may need to be edited to bring their typography into conformance, usually by editing each file individually, because few systems have the ability to apply the same editing operations to each file in a list.

Knock-on viscosity is the other main type, frequently found in structures that have high inter-dependencies, such as **timetables**:

	9.00-10.00	10.00-11.00	11.00-12.00
Year 1	Mr Adams	Ms Burke	Ms Cooke
Year 2	Ms Cooke	Mr Davis	Mr Adams
Year 3	Ms Burke	Mr Adams	Mr Davis

Figure 3 A timetable showing which teacher is teaching which class. Timetables typically contain high knock-on viscosity. If Mr Adams's Year 1 class has to be moved to the 10-11 slot, everything else will have to be re-organised.

In practice, potent combinations of knock-on and repetition viscosity are the type to be feared, turning up frequently in **graphics structures**. Although word-processors are steadily reducing the viscosity problem for document generation, drawing packages are not making comparable progress. Editing a drawing usually requires much tedious work, and frequently many similar alterations need to be made to different parts of the picture; automation tools, desirable as they might be, are not yet commercially available.

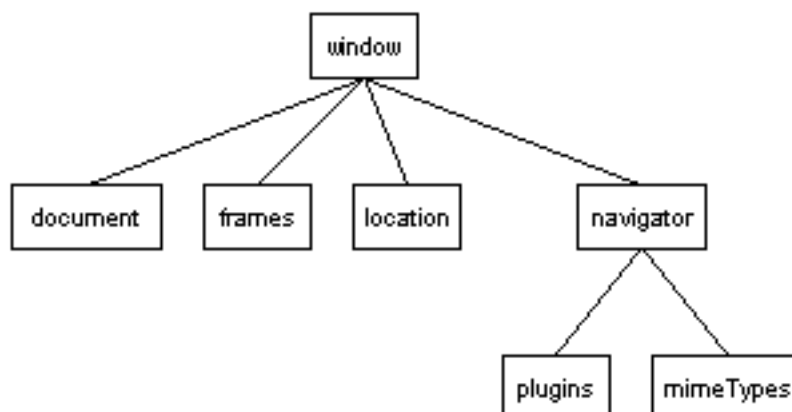


Figure 4 This tree-chart, showing part of the JavaScript object hierarchy, illustrates both knock-on viscosity and potential repetition viscosity: if the 'window' box is moved to one side, all the lines connecting it to other boxes will have to be moved (knock-on). In most drawing systems each line will have to be redrawn individually (repetition viscosity).

Drawing a **genealogical tree** for any reasonably extensive family will well illustrate the difficulties.

Many **programming languages** exhibit pronounced viscosity. This is a serious problem, because software is always changing – during development and also after it is supposed to be finished. Many of the trends in programming language development have therefore been aimed at reducing viscosity. One of the most recent is object-oriented programming, which allows a single change to be made in a parent class, thereby altering the behaviour of many different subclasses. Unfortunately this has the side effect of introducing a new kind of viscosity - if the interface to a class changes, rather than its behaviour, this often results in knock-on changes to its parent class, as well as to all its siblings, and the contexts in which they are invoked (even though they may be only tenuously related to the original change).

So far we have described non-interactive examples, but the viscosity concept is readily extended to interactive systems. Think of using devices as uttering sentences in an interaction language. For non-interactive devices (i.e. those where the names of actions persist and their execution is delayed, such as programs and sheet music) viscosity obviously consists in the work required to reshape the actions. For interactive devices, in which the actions are executed as soon as named (e.g. dialling a telephone, finding a given radio programme) viscosity consists in the work required to get from one state of the device to another state: for the **telephone**, after a slip of the hand has caused a wrong digit to be dialled, the only remedy is to cancel the transaction and restart; for the radio, changing to a different programme may be easy or difficult (see example below).

Other contexts

Hypertext structures, although easy to create, can become extremely viscous. Fischer (1988), describing what appears to be the process of opportunistic design that takes place while creating a collaboratively-generated hypertext structure, writes “ ... For many interesting areas, a structure is not given a priori but evolves dynamically. Because little is known at the beginning, there is almost a constant need for restructuring. [But] despite the fact that in many cases users could think of better structures, they stick to inadequate structures, because the effort to change existing structures is too large.”

Tailorable systems seem to create opportunities for viscosity at the organizational level. Examples of these systems include customisable settings and macros for spreadsheets and word-processors, and other systems allowing users to tailor their office environment. The problem here is that users lose the ability to exchange tools and working environments; moreover, the arrival of a new release of the software may mean that it has to be tailored all over again, possibly with few records of what had been done before.

Indeed, many organizations are repositories of information among many people (‘distributed cognition’) and the organizations themselves can be therefore be considered as information structures. Changing a component of the information held collectively by the organisation may be a matter of telling every single person separately (Repetition Viscosity), or it may, in a hierarchical or bureaucratic organization, only need one person to be told. The levels of the hierarchy have the same role as the abstractions in other systems.

Viscosity growth over time

Finally, diachronic aspects of viscosity. In many **design systems**, the viscosity of any one structure or design tends to increase as it grows. In the context of CAD-E design, one workplace has reported that their practice is to explicitly declare a point after which a design can no longer be changed in any substantial way. At that point the design is said to have ‘congealed’, in their terminology, and opportunistic design must be restricted to details. There is at present no reported data, to my knowledge, describing in detail the time course of viscosity in different systems. The impression is that certain programming languages, such as Basic, have low initial viscosity, but as programs grow their viscosity increases steeply. Pascal starts higher but increases more slowly, because more is done in the initial abstractions. Object-oriented systems are intended to preserve fluidity even better, by using late

binding and the inheritance mechanism to minimise the effort needed to modify a program. Some of the investigations made under the banner of software lifecycle studies may prove to be revealing.

Workarounds, Remedies and Trade-offs

Before considering possible remedies, observe that viscosity is not always harmful. Reduced viscosity may encourage hacking; increased viscosity may encourage deeper reflection and better learning. Reduced viscosity may allow small slips to wreak disaster; for safety-critical systems, high viscosity may be more desirable.

Nevertheless, viscosity is usually to be avoided, and some of the possible manoeuvres are:

- § *The user decouples from the system:* the workaround for unbearable viscosity (especially when combined with a need for early or premature commitment) is to introduce a different medium with lower viscosity to make a draft version, and then transfer it to the original target medium. Thus, the process is now two-stage: an *exploratory* stage followed by a *transcription* stage.

Thus, before drawing in ink, an artist may make a sketch in pencil, which can be erased. A programmer may work out a solution on paper before typing it.

Alternatively, in some circumstances users may decouple by adopting a different notation instead of a different medium. The programmer may choose to develop a solution in a ‘program design language’ before transcribing it into the target language.

- § *Introduce a new abstraction.* This is the classic remedy. Got a problem with updating figure numbers? OK, we’ll create an AutoNumber feature. See below for the trade-off costs of abstractions.
- § *Change the notation,* usually by shifting to a relative information structure instead of an absolute one. Consider a list of quick-dial codes for a **mobile telephone**:

1	Aunt Mary
2	Neighbours
3	Take-away
4	Office
5	Car rescue service

Figure 5 Modifying the order of entries in a telephone memory is usually a high-viscosity operation

I want to be able to dial ‘Car rescue service’ very quickly indeed, let us say. That means moving it to the top of the list, because on this model of mobile phone the dialling codes are scrolled one by one until the required target is visible. I want to move that one item while keeping the others in their existing order. Almost certainly, on existing models, I will have to re-enter the whole list; but in a good **outliner** system, I could perform exactly the same modification in a single action, by selecting the Car rescue entry and dragging it upwards:

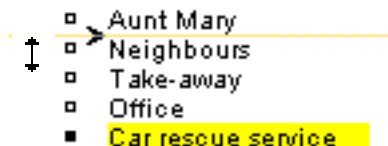


Figure 6 Order modification in an outliner is a low-viscosity operation. The highlighted item (car rescue service) is being dragged upwards; when released, it will take its place where the horizontal arrow points, and all lower items will be displaced downwards.

Many other dimensions can affect viscosity, but it will obviously be easier to discuss such trade-offs when we reach the dimensions concerned.

Hidden Dependencies

Definition

A hidden dependency is a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible. In particular, the one-way pointer, where A points to B but B does not contain a back-pointer to A.

The *search cost* of a hidden dependency structure is a measure of the effort required to expose a typical dependency. Search cost is a function of the length of the trail, the amount of branching, and the effort required to follow each link.

Thumbnail illustration

HTML links: if your page is linked to someone else's, e.g. the HCI 98 site, how will you know if and when that page is moved, changed, or deleted?

Many links are *fossils* – out of date pointers to pages that have been deleted or moved. Because checking links is slow, the *search cost* for testing integrity of a site is quite high, so fossils are likely to increase over the years.

Explanation

A dependency can be hidden in two different ways, by being one-way or by being local. The HTML link is a one-way pointer, so it is invisible from the other end – I cannot find out what links to my page have been created, unless I use a search engine. As it happens, HTML links are also local: they point to the next page on a chain, but say nothing about what lies beyond.

Spreadsheet links are just the same in structure. Here their local nature is more important, because I may well want to know which cells use the value in a given cell, whether directly or indirectly.

Cognitive relevance

Hidden dependencies slow up information finding. If the user wishes to check on a dependency and has to find it by a lengthy search process, the job of building or manipulating an information structure will take much longer; the risk of error will increase; and the train of thought will be broken, adding again to the time and unreliability of the process. When dependencies form long trails and there may be several branches at each step, the job of searching all the trails becomes very onerous, and users are tempted to make guesses and hope.

Exploratory design can tolerate hidden dependencies for small tasks, but they are problematic for modification activity. Transcription activity is probably not much affected except when mistakes are made.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>hidden dependencies</i>	<i>acceptable</i>	<i>acceptable</i>	<i>harmful</i>	<i>acceptable for small tasks</i>

Cost Implications

Hard to estimate, but hidden dependencies are apparently responsible for a very high error frequency in spreadsheets, possibly because the spreadsheet programmer hopes for the best rather than scrupulously checking out all the consequences of a change.

Types and Examples

Dependency pointers can be one-way or symmetric, local or wide, and hidden or explicit.

One-way and symmetric pointers

A **GOTO statement** in a programming language is a one-way dependency, because at its target there is no corresponding COME-FROM. A flowchart also contains GOTOs but they are represented symmetrically; the arrow shows which commands can be the targets of GOTO commands. By using indentation (and restricting the range of control constructs) textual languages can be given symmetric pointers:

```

1:   J = 1                               for J=1 to 10
2:   IF J > 10 GOTO 6                     {
3:   J = J + 1                             compute
4: (compute)                               }
5:   GOTO 2                               (continue)
6: (continue)

```

Figure 7 One-way pointers are shown in the code on left. The code on the right contains implied pointers that are symmetric.

Local and distant pointers

A local pointer only leads to the immediate target; a distant pointer gives a view of the farther reaches of the dependency. A file-structure displayed on a Unix platform by using the 'ls' command shows the children of a given directory, but it does not show the grandchildren, nor the grandparents. The same file-structure displayed as a tree shows the entire structure.

Hidden and explicit pointers

An explicit pointer is one that is shown in the normal viewing state. A hidden pointer is not shown in the normal viewing state, and indeed may not be shown at all, so that it has to be searched for.

Examples

Spreadsheets are a mass of one-way, local dependencies. Even in the formula view, as illustrated, we cannot detect, by looking at B1, that it is used in F2. Some modern spreadsheets contain tools to analyse dependencies, but early versions had none, so that altering a cell in someone else's spreadsheet was a risky business.

The search cost for spreadsheets rises sharply with the size of the sheet and with the branching factor (average number of data-supplying cells quoted, per formula):

	A	B	C	D	E	F
1	item	unit price	units	price	tax	total price
2	apples	£ 0.84	2	=B2 * C2	0.29	=E2 * D2

Figure 8 Spreadsheets illustrate hidden dependencies.

Certain types of **visual programming language** make the data dependencies explicit, as shown, thereby accepting different usability trade-offs:

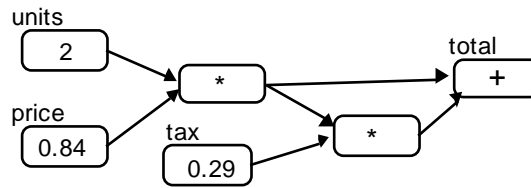


Figure 9 Data flow representations make the dependencies explicit.

Styles, object hierarchies, and other abstractions usually impose hidden dependencies on their users: change a style definition, and who knows what else will change in your document.

Contents lists and similar lists obviously contain hidden dependencies. Every now and again we all forget to update a contents list or its equivalent.

Surprisingly, contemporary **car radios** contain hidden dependencies: they contain a feature for traffic information, so that as you are driving announcements can cut in over the programme you are listening to, to warn of congestion or other problems. These announcements come from a different station, so they may be louder or quieter (or in any case you may have been playing a tape). To avoid getting your ears knocked off, there is a feature for pre-setting the maximum volume. An interesting problem for the user, because pre-setting the maximum volume is (a) a problem in premature commitment, (b) has no chance of progressive evaluation, and (c) is a hidden dependency -- when I'm driving and vaguely listening to a soothing bit of Radio 3 and suddenly a geezer starts banging on at about 100 dB about traffic delays in some place I've never heard of, I want shot of him NOW, not after 30 seconds of fiddling around. The preset is a hidden dependency, because there is no link back from the instrument's present mode to the mode that allows me to change the maximum allowed volume.

Conventional programming languages are built upon hidden dependencies. Assignments made in one statement are then used elsewhere; a simple form that usually causes few problems, because the search cost is not too high (except in the notorious side-effect, where values were surreptitiously altered in unexpected places). Type declarations can sometimes create hidden dependency problems.

Scripting languages have now become common adjuncts of advanced applications. Event-based versions, such as Visual Basic, HyperCard and Toolbook, are notorious for hidden dependencies, because they spawn little packets of code attached to various buttons and other graphics, communicating by shared global variables. Figuring out which packet of code does what when becomes a nightmare. Despite which, they are very popular for some purposes, because the trade-offs are favourable for some types of activity – see below.

Fossil deposits

Fossils, mentioned in connection with broken web links, are becoming a noticeable feature of **operating systems**. Unix platforms use 'dot files' for various purposes, typically to contain preference data for editors and similar utilities; these are hidden dependencies, because various programs may expect to find a particular dot file in place, and crash if it is not there. Users may find that their directories contain a large number of dot files whose purpose is obscure, but which they hesitate to delete in case something does need them. PC platforms under Windows have similar large deposits of fossils, notably out-of-date DLLs.

In general information structures that cause a build-up of fossils are poor choices for long-term use, since they will obviously suffer from 'mature disfluency' – that is, they will get harder to use as time goes on.

Workarounds, Remedies and Trade-offs

§ *Add cues to the notation*: To avoid hidden dependencies in programming, a number of designs have been proposed. Among these are the scrupulously 'neat' languages like the Modula family, in which the technique for encapsulating code into functions or procedures requires very explicit statements of what identifiers are publicly visible through the 'wall'.

§ *Highlight different information:* The dataflow-based visual programming language illustrated above exposes the data dependencies as the central feature of the notation, a different paradigm that accepts an entirely different set of trade-off positions from say, a C version of the same algorithm.

§ *Provide tools:* Modern spreadsheets can highlight all cells that contribute to the value in a particular cell, or alternatively all cells that draw upon the value in a given cell. This has its uses, although it has to be noted that the visibility is reduced. The dependency information for two different cells is not easy to compare, for example.

Do not rush to eliminate hidden dependencies. Among the most popular programmatic creations have been Basic, especially Visual Basic; spreadsheets; and the event-based family of HyperCard, Toolbook, et al. All of these are high on hidden dependencies.

Evidently, hidden dependencies are quite acceptable in appropriate circumstances. The languages mentioned are typically used for quick and dirty coding or for exploratory design. Both strategies mean getting your ideas down quickly. A good designer can accept a high mental load, remembering what bits depend on what, until the main structures are in place, but cannot accept constant fiddly interruptions. So an incremental system, like a spreadsheet, is good for quickly assembling some material and some formulas. During that design phase, because the links are only one-way, the viscosity is lower; if the links were two-way, then every time a new addition was tried and then corrected, some links would need to be adjusted, slowing up the creative process. Architects seem to work the same way.

The problem of exploratory design comes in the later stages, when the structure tries to turn into a congealed mess. Experienced designers probably know how far they can go without cleaning up. For lesser mortals, environments that can document links automatically would be useful.

If hidden dependencies are to be reduced, consider the costs of the remedies. Remedy (i), where the information is made explicit by cueing, as in the Modula family, is probably more suited to a system designed for transcription activity than for exploratory activity. The remedy makes the notation much more viscous. Remedy (ii), where the notation is focused upon the dependencies, creates a more diffuse language, sometimes with its own problems of premature commitment (where shall I draw the next operation?) etc. Remedy (iii) has poor juxtaposability.

Premature Commitment and Enforced Lookahead

Definition

Constraints on the order of doing things force the user to make a decision before the proper information is available.

Thumbnail illustration

The amateur signwriter at work:



Explanation

The problems arise when the target notation contains many internal constraints or dependencies and when the order constraints force the user to make a decision before full information is available (premature commitment) or to look ahead in a way that is cognitively expensive (enforced lookahead).

The amateur signwriter's target notation contains a constraint relating the width of the sign-board and the width of the wording, but until the sign was written he or she did not know how wide the wording would be – so they made a guess (premature commitment). The professional would paint the sign on a different surface, measure it, and copy it to the real target (enforced lookahead, using an auxiliary device).

Cognitive relevance

An obvious disruption to the process of building or changing an information structure.

Problematic in all contexts, except where the lookahead is not extensive. Probably one of the important differences between 'novices' and 'experts' in studies of design is that experts recognise potential problems much earlier, perhaps not from looking ahead but by familiarity with likely sources of difficulty. Research has shown that expert designers frequently treat potential trouble-spots differently, putting them aside for later treatment or attacking them early.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>premature commitment</i>	<i>harmful</i>	<i>harmful</i>	<i>harmful</i>	<i>harmful</i>

Cost Implications

One origin of premature commitment is that the designer's view of a 'natural sequence' was at variance with the user's needs. A second possible origin is that the technical aspects of the design made it necessary to deal with one kind of information before another type, regardless of which was more

salient to the user. Either way, the result is that users will have to make second or even a third attempt. The cost implications will vary from trivial to unacceptable, depending on the circumstances of use.

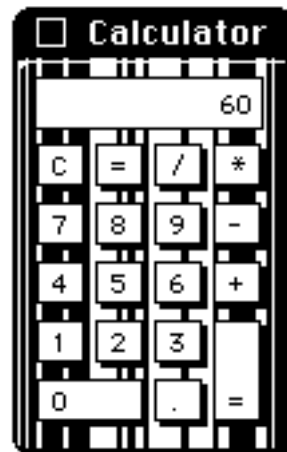
Types and Examples

Telephone menu systems (“If you wish to make a booking, press 1 ...if you wish to reserve a seat, press 2 ... etc”) impose premature commitment on the long-suffering user, who has to decide whether to press key 1 before knowing what other choices are available.

The 4-function **calculator** is another everyday example: What keypresses are needed to compute

$$(1.2 + 3.4 - 5.6) / ((8.7 - 6.5) + (4.3))$$

on the calculator illustrated?



Turning to software examples, every amateur has met the problem of choosing categories for a **database** and then discovering that the world is full of exceptions. For example, my simple mailmerge list of addresses allows 6 lines per address. Now and again I meet somebody whose address needs 7 or more lines. What now? The first design of the database had to be based on guesswork, in other words I was prematurely committed.

In an example that is closely related in its aetiology but far removed in scale, the structure of a traditional (non-computerised) **filing system** is usually susceptible to premature commitment. If the filing system is to hold documents that already exist, the problem is less acute, but if the documents do not yet exist, hard choices have to be made (as they say). Dewey’s decimal classification was a brave effort to categorise the world, whose effectiveness has been seriously marred by the later development of whole new branches of knowledge which were unknown to him.

Old library buildings which have left no space for modern knowledge areas have an even worse problem. Designed before the development of biochemistry or information technology, their science shelves are bursting out all over.

Surreptitious order constraints

But even when the world of possible instances is reasonably well understood, the system can make problems by denying users the opportunity to choose their own order of actions. Consider designing a graphic **editor** for entity-relationship diagrams; most of us would probably provide a facility to define new entities, and a facility to define new relationships between existing entities. The user’s choice of actions appears to be free – entities can be added, deleted, or modified at any time, and relationships between existing entities can likewise be added, deleted, or edited at any time.

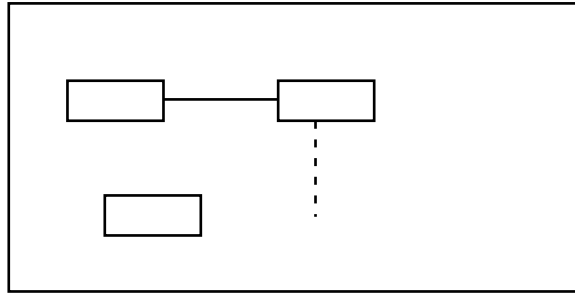


Figure 10 A box-and-line editor (e.g. for entity-relationship diagrams). These editors frequently impose a type of order constraint: users can add boxes with no connectors, but cannot add connectors except between existing boxes. The dashed line is an example of a 'free' connector, that could not be drawn.

Yet that system imposes an order constraint. If the user wants to say “OK, this entity has to have a 1:many relationship with something, but I don’t yet know what”, there is no way to record that fact. The design only permits relationships to be defined between pre-existing entities – thereby making the implementation much easier, no doubt, but imposing on the user the burden of remembering dangling relationships until the associated entities have been defined.

Effect of medium

Premature commitment is more strongly affected than other dimensions by the choice of medium. When ‘marks’ are transient, as with auditory media, it is essential to help the user make good choices by providing ample information. When the ‘marks’ are persistent, as with paper, the user usually has time to make a choice by juxtaposing them and choosing the best. Thus, even when the telephone menu is too complex to be usable, the same information structure could be handled facily and routinely if the medium were paper or some other persistent medium.

Workarounds, Remedies and Trade-offs

- § *Decoupling*: As with viscosity, the obvious workaround is to use an intermediate stage to decouple from the immediate problem. The amateur signwriter should paint the sign somewhere else, measure it, and then allow adequate space.
- § *Ameliorating*: Premature commitment is less costly when viscosity is low, since bad guesses can be easily corrected, so one remedy is to reduce the viscosity in the system.
- § *Deconstrain*: The ideal remedy is to remove constraints on the order of actions. This has been one of the contributions of GUI interfaces.

Abstractions, Abstraction Hunger, and the Abstraction Barrier

Definition

An abstraction is a class of entities, or a grouping of elements to be treated as one entity, either to lower the viscosity or to make the notation more like the user's conceptual structure.

The *abstraction barrier* is determined by the minimum number of new abstractions that must be mastered before using the system; if the system allows one user to add new abstractions that must then be understood by subsequent users, that will further raise the abstraction barrier.

Abstraction-*hungry* systems can only be used by deploying user-defined abstractions. Abstraction-*tolerant* systems permit but do not require user-defined abstractions. Abstraction-*hating* systems do not allow users to define new abstractions (and typically contain few built-in abstractions).

Thumbnail illustrations

Styles in word processors: an example of abstraction-tolerance. The **Z specification language**: an abstraction-hungry system with a high initial abstraction barrier. Spreadsheets: an abstraction-hating system.

Explanation

The characteristic of an abstraction from our point of view here is that it *changes the notation*. Almost invariably, the notation is changed by expansion – a new term is added. In the specification language **Z**, the off-the-shelf notation contains a large number of primitives, which can be assembled to form abstract specifications of parts of a device's behaviour; the abstractions can then be assembled to form a complete specification of the device.

It may be less apparent how abstractions change the notation for interactive devices. Think of interacting with them as uttering 'sentences' in an 'interaction notation'. The sentences for setting all level 1 headings to 24-point bold would include commands to find next heading, select it, set size to 24, and set style to bold. By creating a style called Heading 1, an alternative notation is created – select Heading 1 and set size and style.

In the same way, the off-the-shelf notation for dialling **telephone numbers** comprises the digits 0-9. If the telephone has a memory the notation can be changed to include *1 (meaning dial 0123-456-7890, say), and so on.

Cognitive relevance

Abstractions make notations more concise and sometimes they make them fit the domain concepts better. They are an investment in the future: by changing the notation now, it will be easier to use in the future, or it will be more re-usable, or it will be easier to modify what's built in the notation because the abstractions can reduce the viscosity.

Using abstractions is costly. First, they are hard. Thinking in abstract terms is difficult: it comes late to children, it comes late to adults as they learn a new domain of knowledge, and it comes late within any given discipline. Systems that enforce abstractions will be difficult to learn and may permanently exclude some possible users. (We can see why they are hard from machine learning theory, where the constant problem is to avoid either over-generalisation or over-specialisation.)

Second, they must be maintained (see the section on abstraction management). Creating and editing them takes time and effort and has potential for mistakes.

Thus, abstractions are potentially useful for modification and transcription tasks. They are good for incrementation tasks if they fit the domain well. For exploratory tasks, they are problematic. Highly skilled designers can use them, but they are most likely to be *exploring the abstractions needed* as well as exploring for a solution given the abstractions. Less skilled designers working with smaller problems typically prefer to save the investment, even at the expense of using a more long-winded notation and building structures that will be more viscous.

That is equally true of interactive and non-interactive systems. Less skilled word-processor users will avoid styles, auto-formatting, macros, advanced (pattern-matching) editing, and other forms of abstraction. These features could save effort, but they are error-prone and cognitively costly.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>abstraction hunger</i>	<i>useful</i>	<i>useful (?)</i>	<i>useful</i>	<i>harmful</i>

Cost Implications

Programming languages often use many abstractions (procedures, data structures, modules, etc) although spreadsheets use rather few. Everyday domestic devices tend to use rather few abstractions, although telephones with built-in memory systems have a form of abstraction, some central heating controllers use simple abstractions (for weekdays/weekends, holidays, etc), cars can or soon will be ‘personalised’ so that each driver can have the adjustments for seating, mirrors, radio stations etc. adjusted individually at one go, and so on

Abstractions have a key role in the future of information devices, because they directly govern many aspects of ease and because they require abstraction managers, which themselves are sub-devices with their own usability characteristics.

Abstractions are proliferating, especially in personalised and embedded computers. The next generation of car controls, including seat adjustments, in-car audio systems, etc., will include advanced **personalisation** features: tell the car who you are (in any of various ways) and the seats and mirrors will be adjusted, your own set of radio stations and preferences will become available, and so on. The ‘person’ is a new abstraction in this area. **Digital broadcasting** will bring many other new abstractions. We shall repeatedly see tensions between the effort of learning to cope, and the convenience when the system has been mastered; we shall have problems of preserving and exporting the definition of ‘Thomas’ or ‘Alan’, when we get a new car; we shall be frustrated by the overhead of having to modify definitions as our personal characteristics change over time; and it may become almost impossible to borrow a friend’s car in a hurry.

It is worrying that very little research has looked at the learnability, usability, and maintenance issues. The difficult thing about abstractions is trying to get the scope right. An abstraction that describes a particular set of items is not very useful, because it cannot be used for other similar items in the future. This is an over-specialised abstraction. An abstraction that describes all possible items is not useful either, because there is almost no action that one would want to specify for all possible items. This is an over-generalised abstraction. Good abstractions depend on a well-chosen set of examples, and they also depend on you knowing what you want to leave out. You often don't know what you want to leave out at the time you encounter an abstraction barrier, and this results either in confusion or in bad abstractions

Abstraction managers

If new abstractions can be created, some kind of sub-device must be provided to manage the user-defined abstractions. This sub-device may or may not suffer from viscosity (it is easy to change an abstraction); from hidden dependencies (will changing one abstraction impinge on others); juxtaposability (when I’m editing one abstraction, can I compare it others); etc.

Types and Examples

Since abstractions operate to change the notation, they have second-order effects with a potential to vastly increase existing problems. They come in several types.

Persistence

Persistent abstractions, as their name implies, remain available until explicitly destroyed. *Transient* abstractions automatically cease to exist, usually after one use.

Style sheets, macros, object hierarchies, grouping in drawing applications, and **telephone memories** are persistent abstractions.

Transient abstractions include **aggregates** selected by pointing, e.g.:

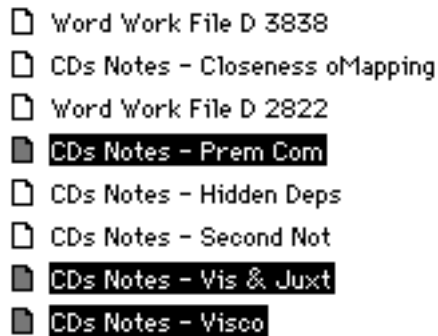


Figure 11 A transient abstraction. Three files in a MacOS folder have been selected by point-and-click, forming an aggregate that can be trashed, printed, opened, or copied as one unit. This aggregate is transient, because the next MacOS command will deselect the files.

Transient abstractions also include **global find-and-replace** commands, e.g.

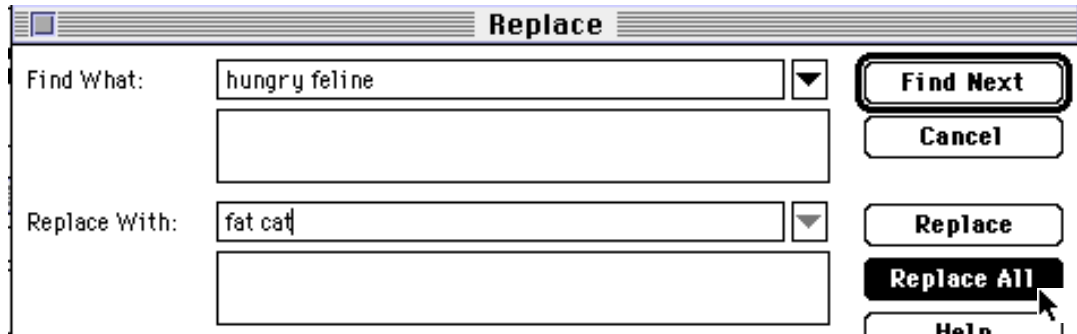


Figure 12 Another transient abstraction. This search-and-replace command in Word will change all hungry felines to fat cats.

Aggregates, definitions, and exemplars

Abstractions may aggregate several entities into a single unit, as the selected folders do, or they may aggregate several actions into a **macro** definition. These different contents have rather different properties (you can remove one entity from an aggregate of entities without affecting the rest: but just try removing one action from a macro and see what you get!), but for broad-brush purposes we shall ignore the differences.

Abstractions may also be defined as exemplars, as in many **CAD** programs where a sub-structure can be encapsulated and stored in a library to be re-used as required. This allows further sub-varieties: if the exemplar is changed, will all existing copies of it be updated? Or will the change only affect new copies?

Word styles are automatic-update abstractions. Redefining one automatically changes all existing instances of that style in that document (together with instances of other styles that inherit properties

from it). Note the potential for hidden dependency here – if we cannot readily see which examples will be changed, we may be in for nasty surprises.

Powerpoint templates, used to define the look of a presentation, are no-update abstractions. If the template is changed, existing presentations are not automatically updated.

Sometimes, optional-update abstractions are available.

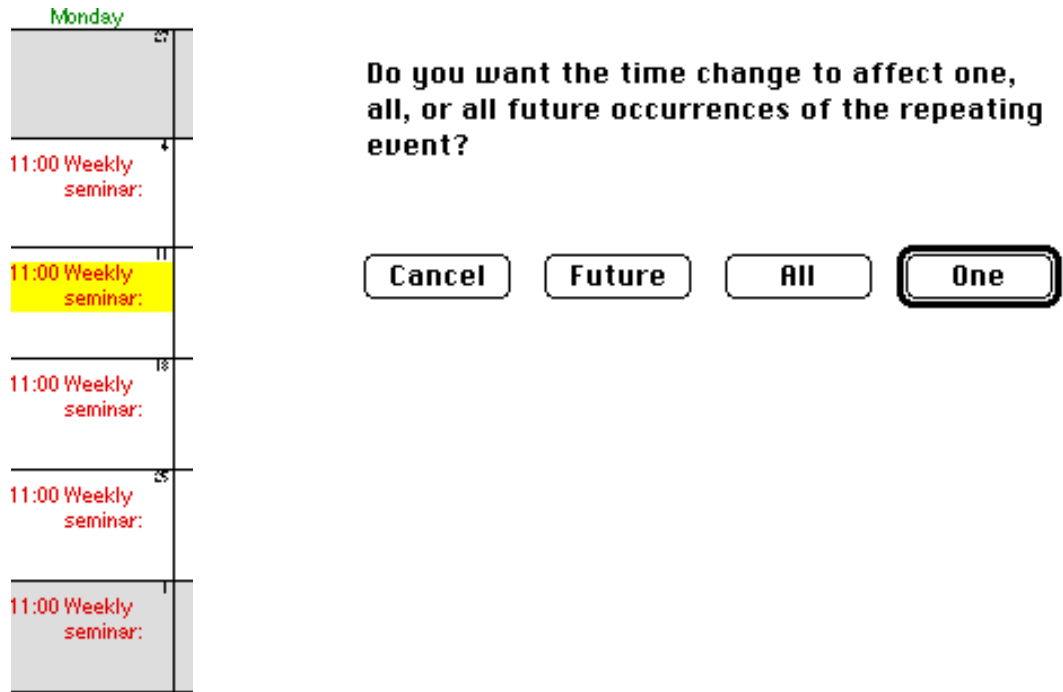


Figure 13 Optional update in a diary system: the time of one instance of the weekly seminar is to be altered. A dialog gives options of which other instances to update.

Abstraction hunger and the abstraction barrier

Flowcharts are abstraction-hating. They only contain decision boxes and action boxes. **Spreadsheets** are more various, but the elementary spreadsheet has no facilities for abstractions.

C and **HyperTalk** are both abstraction-tolerant, permitting themselves to be used exactly as they come, but also allowing new abstractions of several kinds to be created. HyperTalk has a low initial level of abstractions (i.e. a low abstraction barrier), and novices can start by learning how to use fields, then how to use invisible fields, then how to use variables. C has a much higher minimum starting level, so that the novice has many more new ideas to learn before being able to write much of a program.

Smalltalk has a high *abstraction barrier* because many classes must be understood before using it, and moreover it is an *abstraction-hungry* language: to start writing a program, you first modify the class hierarchy. The user must feed it new abstractions. This is a whole new ball-game. Every program requires a new abstraction; the novice programmer cannot simply map problem entities onto domain entities. (Of course, experienced programmers have a stock of abstractions they have already learnt, ready to be applied in a new situation.)

The abstraction management problem has been mentioned above. By and large, abstraction management subdevices are designed thoughtlessly and poorly. For instance, Word allows users to define new styles, a very useful abstraction. The styles are held in a hierarchy, each inheriting all the characteristics of its parent except those that explicitly over-ridden.

Two examples of problems with the Word 5.1 style manager: hidden dependencies and lack of juxtaposability. (i) Modifying a style will potentially modify all its children – i.e. there are dependencies – but these dependencies are completely hidden: the Modification dialog box shows the parent of a style, but not its children, thus:

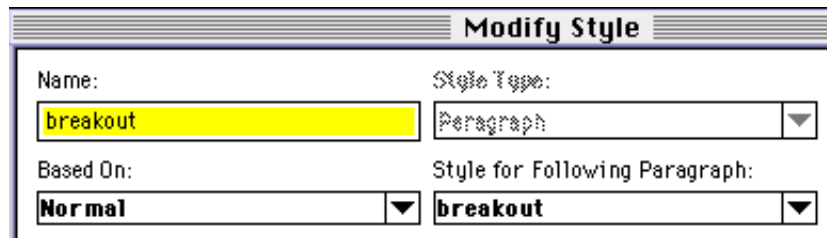


Figure 14 Hidden dependencies in the Word 5.1 style manager: if I modify the definition of this style, I cannot tell which other styles will be affected.

(ii) The lack of juxtaposability is imposed by the same dialog box. The modification box illustrates the style, but I cannot open two modification boxes at the same time. In consequence, almost-identical styles proliferate.

Workarounds, Remedies and Trade-offs

§ *Incremental abstractions*: the best workaround is seemingly to provide users with a system that has a low abstraction barrier but yet contains useful built-in abstractions as short-cuts, and tolerates the addition of useful new abstractions. Novices can then start by doing operations the long but conceptually simple way, and master alternative methods later.

The limits of this come when the alternatives create a morass of bewilderment. The ‘latest, fully-featured applications’ are always to be distrusted on this score.

§ *Trade-offs from abstractions*: abstractions are a standard way to reduce viscosity. They can also increase the protection against error-proneness (for example, by declaring all identifiers, mistypings can be detected at compilation rather than run-time). Well-chosen abstractions can also increase the comprehensibility of a language. But remember that the abstractions may have to be chosen almost by guesswork, so that premature commitment may be substantial and may not be easy to undo.

But abstractions will increase the problems for occasional users and for learners. Learning how to work with abstractions is difficult, and understanding a structure that contains many abstractions is often difficult. Moreover, abstraction-hungry systems necessarily take a long time to set up, suffer from a sort of delayed gratification, because the appropriate abstractions must be defined before the user’s immediate goals can be attacked.

We have already mentioned the problems of hidden dependencies and juxtaposability that bedevil so many abstraction managers.

One reason why spreadsheets are so popular is that they are weak on abstractions. In fact, not surprisingly, many potential end-users are repelled by abstraction-hungry systems.

One proposed workaround for the abstraction barrier is **Programming By Example**, in which you show the system some items, and it decides what the abstraction should be. This tantalising idea doesn't seem to have worked very well so far. The errors of excessively over-general or over-specialised abstraction are even more likely to be committed by a machine with no recourse to common-sense sanity checks. Even worse, you still have to do the hard work – identifying the set of examples (including appropriate negative examples) that will be used to define the abstraction. As a programming method, programming by example has the potential to introduce some horrendous problems combining hidden dependencies and premature commitment.

Secondary Notation

Definition

Extra information carried by other means than the official syntax.

Redundant recoding gives a separate and easier channel for information that is already present in the official syntax. *Escape from formalism* allows extra information to be added, not present in the official syntax.

Thumbnail illustration

Indentation in programs (redundant recoding); grouping of control knobs by function (redundant recoding); annotation on diagrams (escape from formalism).

Explanation

The formal notation can sometimes be supplemented with informal additions which have no official meaning, but which can be used to convey extra information to readers. Indentation or ‘pretty-printing’ is an essential part of a programming language, but it has no meaning to the compiler in most cases: it is a form of secondary notation. Exactly the same pretty-printing idea is used to help make telephone numbers easier to read, although interestingly enough, conventions differ from country to country - in Britain we split numbers into threes and fours, but in France they split them into two-digit groups.

Cognitive relevance

Redundant recoding makes comprehension easier. Easier comprehension makes easier construction: all research on the design process shows that creating information structures is a looping process in which the part-finished structure is inspected and re-interpreted.

Highly relevant to exploratory and modification activities. Less relevant to transcription activities – but not to be ignored there, either;

Escape from formalism adds to the information available; typical comments will describe the origin, the purpose, the design rationale, the reliability status (e.g., whether tested), or the intended future (‘still to be revised’). Such information is an essential commodity in the process of building or modifying an information structure.

Secondary notation has an equally valid place in interactive systems; a control system in which functionally-related controls are grouped together is likely to be easier to learn – to induce fewer slips of action too.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>secondary notation</i>	<i>useful (?)</i>	–	<i>v. useful</i>	<i>harmful ?</i>

Cost Implications

It is hard for me to imagine any situation that would not be improved by making a notation easier to read, but die-hard C/predicate calculus/whatever users maintain that anyone can understand their pet formalism. Attempts to create less impenetrable notations, such as visual programming languages, are ‘infantilisation of computer science’ (Dijkstra). (If you meet such a person, offer them an exam question in C++ and ask whether they mind if you turn the lights right out and the music full up.)

Unfortunately designers often forget it. It is easy to work with implicit model of users in which they only know, and only need to know, that information which is contained in the official syntax; or to expect users to master all controls, however arbitrary and complicated.

Types and Examples

Redundant recoding

A typical Sheffield **telephone number** is printed as 0114 212 3456, indicating city, area (or service provider), and individual number. Before the Sheffield numbers were changed to include an extra 2, it would have been printed as 0114 123 456. When the extra 2 was added, there was some degree of confusion: was the new digit part of the city code (giving 01142) or not? If Jez Rowbotham believes the Sheffield code is 01142, and if Jez has been told the number as ‘Sheffield 212 3456, what will happen when he tries to ring that number? He will dial 01142 225 533(5) – the final digit being ignored by the telephone exchange. Whoever lives at 0114 222 5533 will get a wrong number.

(At home, TG’s number is frequently rung by people who think we are a ski travel firm and try to arrange ski-ing holidays. “Can I speak to Maureen please?” – Sorry, we’re not the ski-travel firm. “Well, who are you then? I rung the right number, didn’t I? Leeds 234 4321?” – Well, I think you probably rang an extra 2, so you’ve got through to Leeds 223 4432. The code for Leeds is 0113, not 01132. “Well, I dunno, that’s poor role-expressiveness, that is, they oughta improve the secondary notation, why don’t they learn about cognitive dimensions? Sorry to trouble you, bye now”, and so on.)

You can get some idea of the importance of secondary notation in this context by starting to give British people telephone numbers expressed in the French style:

0114 225 5335 becomes 0 11 42 25 53 35

At the same time, do the opposite with your French friends. You will find that your friends are perplexed and probably irritated, because they cannot parse the new presentation into the units they are used to.

The facade or **front panel** of an instrument (e.g. a car radio/cassette) gives designers a chance to use secondary notation to good effect, by grouping related controls in the same area. This is simple enough for an instrument with few facilities (see the telephone example on the web site) but for more complex instruments it can be much harder to bring off well.

Functional grouping is likewise an important aspect of **circuit notations** (Figure 15). The wiring diagram showing the layout of physical components is constrained by factors of size, heat emission and so on, so that components have to be fitted together as best they can be; to make the functional relationships clear, it is usual to present a ‘schematic’ showing the connections between idealised components.

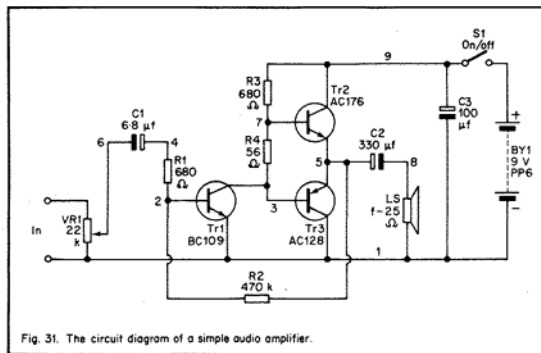


Fig. 31. The circuit diagram of a simple audio amplifier.

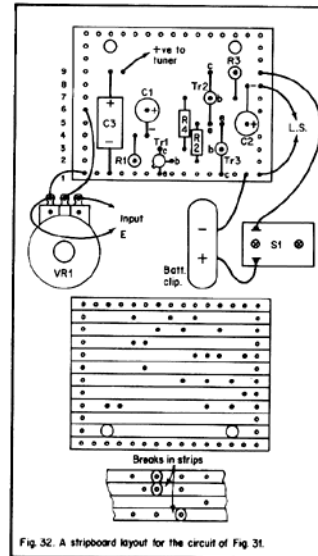


Fig. 32. A stripboard layout for the circuit of Fig. 31.

Figure 15 Secondary notation in circuit diagrams: the left hand shows a schematic in which components have been organised to make their function apparent (redundant recoding). The right hand diagram shows the layout on the circuit board.

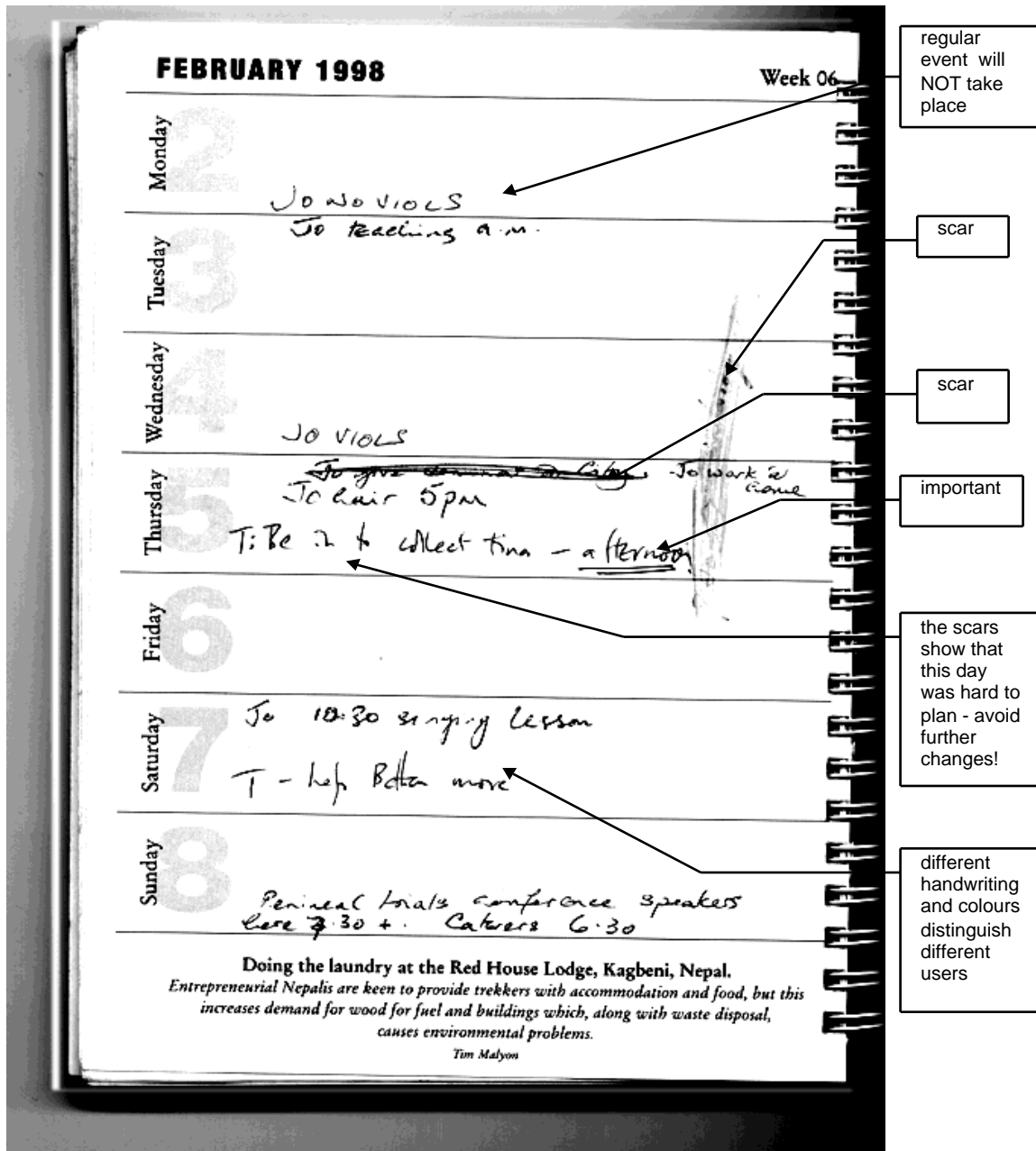
The role of secondary notation is often ignored by the proponents of new **formalisms** and other designs. In the traditional languages, like Basic, redundant information is routinely carried by other means than the formal syntax. Sometimes there is even a ‘visual rhyme’, with similar fragments laid out in similar fashion:

$$\begin{aligned} V_{\text{accel}} &= \text{Thrust} * \text{COS}(\text{Angle}) / \text{Mass} - \text{Gravity} \\ V_{\text{veloc}} &= V_{\text{veloc}} + V_{\text{accel}} \\ V_{\text{dist}} &= V_{\text{dist}} + V_{\text{veloc}} \\ \\ H_{\text{accel}} &= \text{Thrust} * \text{SIN}(\text{Angle}) / \text{Mass} \\ H_{\text{veloc}} &= H_{\text{veloc}} + H_{\text{accel}} \\ H_{\text{dist}} &= H_{\text{dist}} + H_{\text{veloc}} \end{aligned}$$

Note the use of both white space and choice of statement order to get the desired effect. This allows the reader to check and see that the differences are just as expected – i.e. that the vertical component allows for gravity, unlike the horizontal.

Escape from formalism

Now we turn to the use of secondary notation to carry additional information. A study on electronic and paper **calendar** usage (Payne, 1993) showed that paper diaries had an advantage because their users could use big letters to show that something was important, or could draw lines between entries. The electronic forms presented all entries in the same font and had no facilities for adding graphics, so they were unable to express enough information, even though the ‘official’ information was the same. (See figure below.)



February 1998							offline
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	
	Jo - NO VIOLS:	9:00 Jo teaching:	8:00 Jo viols:	pm - T: be in to collect 'tins: 5:00 Jo hair:		T - help Bethan move: 10:00 Jo singing lesson:	

Figure 16 Secondary notation in the paper-based diary (above) can be rich and expressive; much less is available in a computer-based diary (below). In hand-held electronic organisers, possibilities are even more limited.

The programming equivalent is the **comment**. Programmers need to map the program onto the domain and 'tell a story' about it; and they need to document sources ("The next part follows the method of X and Y") or doubts and intentions ("This part is untested, but I'll check it out soon"). Amazingly, many notations intended for non-expert end-users, such as many visual programming languages and even spreadsheets, have little or no provision for comments.

Workarounds, Remedies and Trade-offs

The main workarounds/remedies are:

- ⌘ *Decoupling*: print out a hard copy and attack it with a pencil.
- ⌘ *Enriched resources*: provide tools in the system that allow components to be labelled and described and their relationships made explicit. Hendry and Green (1993) describe a technique for doing this with spreadsheets, called 'CogMap', the cognitive mapper.

Extensive secondary notation creates additional problems of viscosity; if the structure is changed, the secondary notation goes out of date. If the environment contains tools to maintain its accuracy, such as indentation facilities, well and good; but it is hard to imagine tools to support escape from formalism, by its very nature.

Visibility & Juxtaposability

Definition

Visibility: ability to view components easily.

Juxtaposability: ability to place any two components side by side.

Thumbnail illustrations

Visibility: searching a telephone directory for the name of the subscriber who has a specified telephone number.

Juxtaposability: trying to compare two statistical graphs on different pages of a book.

Explanation

In the **telephone number** case, the indexing facilities of a system do not provide the necessary information, the design assumption being that users will rarely want to know which person is number XX. As a real life case that is not actually true: in the UK there is a useful 'who-rang-this-number-last' feature, but sometimes one has no idea about the identity of a mystery number; and in the telephone memory there may be numbers with no attached names, so that one would like to know who they are or which one is the Car Rescue Service, preferably without having to ring each number to find out.

Visibility problems also turn up when the trail of steps leading to a target is longer than desirable, making navigation onerous and possibly error-prone; seeking a particular page on the web, for example (exacerbated here by the typical slow response of the web). In these cases, users building or modifying or querying an information structure are frustrated by the extra work.

Many systems restrict viewing to one component at a time, losing juxtaposability. Typically these systems present information in a window, and are restricted in how many windows can be viewed simultaneously.

Cognitive relevance

The relevance of visibility is immediately apparent in the simple case of searching for a single item of information. Less apparent is that a complex structure is not grasped by inspecting one detail at a time, but by scanning a large sweep (for example, programmers do not read a program one line at a time, but by scanning to and fro). Poor visibility inhibits scanning, a fact too often overlooked.

The relevance of juxtaposability is even more often overlooked. Grasping the difference between two similar items needs close comparison, as in the example of comparing graphs in a book. Consistent treatment of similar data can only be achieved by making sure the second instance is treated the same way as the first. And, more subtly but just as important, the solution to a problem is often found by finding a solution to a similar problem and then building a modified version; or the new solution is compared to an old one to make sure nothing has been left out. The absence of juxtaposed, side-by-side viewing amounts to a psychological claim that every problem is solved independently of all other problems.

Transcription and incrementation only require visibility and juxtaposability for error-checking. Modification makes heavy demands on juxtaposability, to ensure consistency of treatment. Exploratory design uses juxtaposition as a seed for problem-solving.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
<i>visibility / juxtaposability</i>	<i>not vital</i>	<i>not vital</i>	<i>important</i>	<i>important</i>

Cost Implications

Mobile computers and wearable computers are going to make us all too familiar with tiny windows in which juxtaposability has been sacrificed.

Types and Examples

Visibility

Paper-based artefacts, with some centuries of typographical development behind them, are quite good for many purposes, but when the information load becomes too heavy visibility can drop dramatically. Consulting a nation-wide railway timetable will quickly illustrate the problems.

Contemporary designs of domestic information-based devices face a recurrent difficulty that will be familiar to many. These devices allow many parameters to be changed, and it is necessary to provide default settings for some or all of them. **In-car radios** need to have a default initial volume setting for normal programs, and a separate default initial volume setting for traffic announcements (because these are broadcast from a different station, they might otherwise be much louder). **Fax machines** need to store information about user name, the call-back number, the time and date, etc. **Cameras** are going the same way. To access the default settings on these devices, the typical method is to provide a two-level menu system in which the top level covers major groups of functions and the second level provides more detailed items within each group.

The **menu structure** only requires one display item at a time, plus two keys to scroll to and fro through the menu and one further key to ‘accept’ the current item, so this approach avoids both a clutter of information on the display, and a proliferation of special purpose keys. The trade-off is the poor visibility. Confirming that the fax machine’s call-back number or the car radio’s initial traffic announcement volume is correctly set up takes several actions, enough to discourage many users.

To check the call-back number on the British Telecom ‘Image 350’ fax machine, perfectly typical of the menu approach (but rather nicely designed and accompanied by good documentation), the following steps are required (quoted from the instruction manual):

- Press the SET UP button.
- Press the < or > buttons until the display shows USER SET-UP.
- Press the √ (tick-mark) button; display shows LINE SET-UP.
- Press the < or > buttons until the display shows USER DATA.
- Press the √ button; display shows SENDER ID.
- Press the < or > buttons until the display shows CALL BACK TEL NO.
- Press the √ button; display shows the currently set call back number [and allows it to be edited].

The precise number of moves will depend on the state the machine is in, but can go to a dozen or more button-presses.

Figure 17 Visibility problems in a typical personal information artefact mean that checking stored values is slow and cognitively costly.

Just as important, *the user has to remember how to get to the right part of the control tree*. The fax machine described is provided with quite sensible, meaningful item labels; but **in-car radios**, typically, have much terser and more opaque labels, such as DSC (that’s where the default volume settings are – not an

easy guess for the uninitiated: it means ‘Direct Software Control’) or TIM (nothing to do with time – that one means ‘Traffic Information Memo’).

Interestingly, the usual documentation of these devices gives a second-level example of visibility problems. Typically the description is purely procedural, like the one above, which means that to find the information needed the user has to trawl through many pages of very similar procedures. It would be very straightforward to present the menu structure directly, providing a more memorable representation that is more quickly searched. For the fax machine mentioned above, the tree would start like this

```
set resolution
  normal
  fine
  superfine
  photo
set contrast
  normal
  light original
  dark original
print-outs
  (sub-menu items)
directory set-up
(etc)
```

Event-based **scripting languages** create well-known visibility problems. For example, Visual Basic disperses little packets of code (scripts or methods) in many different places through the stack. For a single script, the local visibility is good; the code is usually associated closely with the object that uses it. But the result of the dispersal is that overall visibility is low. Extraordinary debugging difficulties were chronicled by Eisenstadt (1993), caused by the many hidden dependencies and the poor visibility.

Juxtaposability

Problems with juxtaposability are more pervasive than simple visibility problems.

The old-fashioned **alarm clock** dial simultaneously shows current time and alarm time, with a different knob to control each one. Very simple. The digital version usually has two modes, one showing current time, the other showing alarm time, so that the user cannot compare them. Not too much of a problem, but a mild nuisance, and typical of failure to appreciate the need for juxtaposition of information.

Entering data into a **form-based system** is a frequent source of problems; the user wants to make sure that the layout is consistent from one to the next, yet there is no way to see an earlier entry while adding a new entry. The EPSRC Electronic Proposal Form¹ (Version 3.0, as of Feb. 1998) is a splendid example. One section requires names of, and some information about, co-investigators; a subsequent section requires further information about each co-investigator, to be given in the same order – *yet you can no longer see the previous section.*

¹ Available at <http://www.epsrc.ac.uk/resgrant/resform/index.htm>

EPSRC EPS(ERP)

FORM Mills & Boon:Utilities:eps_erp Folder:Visibility demo

Personal Information

The EPSRC aims to encourage equal opportunities. If you are willing to so, please provide information on your own and your colleagues' age, sex and ethnic origin. We will **NOT** use this information in the assessment of this research proposal, but only for internal and statistical purposes.

Please give details for each investigator below.

Date of birth	Gender	Ethnic origin

Figure 18 Lack of juxtaposability in a forms-based system. In a previous section, I entered the names of two proposed co-investigators, A. Einstein and C. Darwin. Now I have to enter personal information about those people. Since there is no link by name, the information has to maintain the order in which the names were entered – either Einstein first or Darwin first – but I am not allowed to see the previous form.

The EPSRC form is clearly designed with transcription activity in mind: the user is expected to have sorted out all the information neatly before starting to enter it. Real life is different. The ‘personal information’ is gained from scraps of paper, old e-mails to be found somewhere in the file-system, or hasty telephone calls, all the while rushing to get the work completed before the mail goes (or some other deadline). Transcription activity has no place in this scenario of opportunistic incrementation, which is in fact more like exploratory design than relaxed transcription.

Often, the question may not be whether consistency has been maintained, but whether information has already been entered. What entries have been made in the memories of a mobile telephone? Do I need to add Aunt Sheila, or is she there already?

Spreadsheets, with their two layers (data and formulas), have asymmetric visibility and juxtaposability. When the sheet as a whole is displaying data, only the formula in one cell can be made visible at any one time. Obviously it will be difficult to compare that formula to any others. When the sheet is displaying formulas, the formula comparisons are easy – but now it is no longer possible to see the values computed by the formulas, which loses much of the point of the spreadsheet model.

The advent of windowing systems brought great advances in juxtaposability for all jobs where more than one task was performed concurrently. A notable improvement for many users was being able to run terminal emulations with windows, using one window for each thread under say X-windows and making a whole new way of working possible; in contrast, on the standard Unix platform without windows, all the threads had to be merged into one stream, making multiple concurrent sessions extremely hard if not impossible to manage.

Workarounds, Remedies and Trade-offs

- § *Working Memory*: When side-by-side viewing is not supported by the environment, users have two choices. The first is to rely on working memory, frequently refreshed by revisiting the two items being compared in turn.
- § *External Memory*: Or else they can make a hard copy of one component to study. This second strategy amounts to *decoupling* themselves from the given environment and proving themselves with a new environment in which side-by-side viewing is possible.
- § *Adding a browser*: Serious visibility problems can be remedied by the provision of browsers, such as Smalltalk-80's class browser, and the provision of alternative views — for instance, some of HyperCard's problems have been eased in other systems by providing a 'contents list' of all the cards in the stack.

Visibility trades off against clutter, which has not been discussed here, and against abstraction level, since providing a browser for an alternative view is creating a new abstraction.

The Remaining Dimensions

The remaining dimensions are more psychological in character. We shall do no more than sketch them here.

Closeness of Mapping

Definition

Closeness of representation to domain

Thumbnail illustrations:

A *close* mapping: the visual programming language LabVIEW, designed for use by electronics engineers, is closely modelled on an actual circuit diagram, minimising the number of new concepts that need be learnt. A *distant* mapping: in the first version of Microsoft Word, the only way to count the characters in a file was to save the file to disc – whereupon it told you how long the file was.

Consistency

Definition:

similar semantics are expressed in similar syntactic forms

Thumbnail illustration:

Many domestic information artefacts are now menu-driven, such as in-car audio sets, with keys to move through the menus. In a consistent version, the same keys move up or down, step to the next sibling, select the current item, or return to the menu display. But frequently there are slight differences from item to item.

Note: Consistency usually affects learnability rather than usability (but see error-proneness). The work of Cramer (1990) suggests that its influence is less than that of mnemonic labelling, which is not considered here because it is not a structural property.

Diffuseness

Definition

verbosity of language

Thumbnail illustration:

COBOL is a verbose language:

MULTIPLY A BY B GIVING C

Forth is a terse language: the command to print a newline is

(i.e. a single full stop).

Cognitive importance:

The effect of over-verbosity is probably slight in most situations, but it should be kept in mind that studies on working memory have shown that mental arithmetic performance is better for children whose native language has short names for numbers rather than long names: the implication is that having to keep long-winded descriptions in mind will limit users who are engaged in activities that demand working memory, such as exploratory design.

Terseness can also be a problem, because it makes it easier to make mistakes (see error-proneness) – an awkward trade-off.

Error-proneness

Definition

notation invites mistakes

Thumbnail illustrations:

(i) Poor discriminability: the language Forth uses full stop and comma to mean different things; Fortran uses the identifiers I and O, regularly confused with one and zero.

(ii) Inadequate syntax-checking: Prolog has no declarations of identifiers, so mistypings cause problems detectable only at run-time.

(iii) Bad dialogue design: inconsistencies in dialogues invite errors, e.g. always using ENTER as a default except in one case.

(iv) Memory overload (caused by premature commitment, etc).

Cognitive importance:

Obvious.

Hard mental operations

Definition

high demand on cognitive resources

Thumbnail illustration:

Threading a maze – new branches to remember are spawned at every choice point. For certain physical mazes there are memoryless algorithms, but following circuit diagrams, auditing spreadsheets, and even finding files in big directory structures are maze isomorphs and are correspondingly hard.

Cognitive importance:

Obvious.

Progressive evaluation

Definition

work-to-date can be checked at any time

Thumbnail illustration

Spreadsheets recompute at every opportunity, so that the user can develop a solution bit by bit.

Cognitive importance:

Novices need frequent checks on work-to-date. Even experienced users will need checks during difficult times, such as working under stress of frequent interruptions.

Provisionality

Definition

degree of commitment to actions or marks

Thumbnail illustration:

Pencils are used by architects, typographers and other designers to make faint blurry marks, meaning ‘something more or less like this goes more or less here’, as well as precise hard marks.

Cognitive importance:

Reduces premature commitment.

Role-expressiveness

Definition

the purpose of a component (or an action or a symbol) is readily inferred

Thumbnail illustration:

This dimension describes the ease with which the notation can be broken into its component parts, when being read, and the ease of picking out the relationships between those parts. An experienced electrical engineer can look at a radio circuit diagram and quickly pick out the parts that deal with different stages of the process (detecting the signal, first amplification stage, etc.).

Cognitive importance:

Although some notations and devices seem to give less trouble, not much research has been reported, and this remains one of the hardest dimensions to clarify, needing further research on parsing and comprehension.

Cognitive Relevance, Trade-offs, and Design Manoeuvres

Each cognitive dimension above has been characterised as particularly relevant to certain user activities. Some of the trade-off relationships and some of the classic design manoeuvres have also been noted. This section tabulates what has been said above.

Cognitive relevance

Here we collect the comments made under each dimension. The columns give **desirable profiles** of dimensions for each of the types of user activity mentioned above.

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
viscosity	acceptable	acceptable	harmful	harmful
hidden dependencies	acceptable	acceptable	harmful	acceptable for small tasks
premature commitment	harmful	harmful	harmful	harmful
abstraction barrier	harmful	harmful	harmful	harmful
abstraction hunger	useful	useful (?)	useful	harmful
secondary notation	useful (?)	–	v. useful	v. harmful
visibility / juxtaposability	not vital	not vital	important	important

Figure 19 Table of user activities

Notable Trade-offs

The issue of trade-offs is very complex. Only some of the more obvious ones have been listed here, especially those that form part of typical design manoeuvres, such as exchanging viscosity for abstractions. Very little research has been reported, to our knowledge, on these issues. The following diagram illustrates the relationships that have been mentioned above.

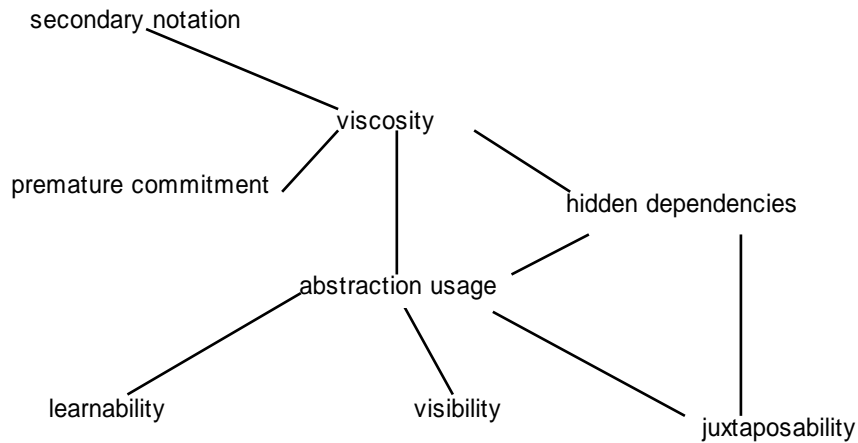


Figure 20 A line indicates a trade-off relationship identified in the text above. Not all of these trade-offs are available in all cases, and the diagram does not pretend to identify all possible relationships.

Design Manoeuvres

Some of the classic design manoeuvres identified above are listed in the following table.

AIM	MANOEUVRE	AT THIS COST
to reduce viscosity:	add abstractions (so that one 'power command' can change many instances)	increases need for lookahead (to get the right abstractions); raises the abstraction barrier; may also increase hidden dependencies among the abstractions
to improve comprehensibility:	allow secondary notation – let users choose where to place, things, how to use white space, what font and colour to use; allow commenting	increases viscosity (because layout, colour etc are not usually well catered for by the environment)
to make premature commitment less expensive:	reduce viscosity (so that users can easily correct their first guess)	see above, re viscosity
to remove need for lookahead:	remove internal dependencies in the notation; or allow users to choose an easier order to make decisions	may make notation diffuse, or increase errors allow free order needs a cleverer system
to improve visibility:	add abstractions (so that the notation becomes less diffuse)	see above re abstractions

Further developments in cognitive dimensions

Although the cognitive dimensions framework succeeds in its aim of being broad-brush, it pays by being too unspecific, making it hard to metricate the dimensions or even to define exactly what they mean. Several projects have set out to improve the situation.

Metrication

Yang et al. (1997) offer benchmarks for 'navigable static representations'. These authors 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)

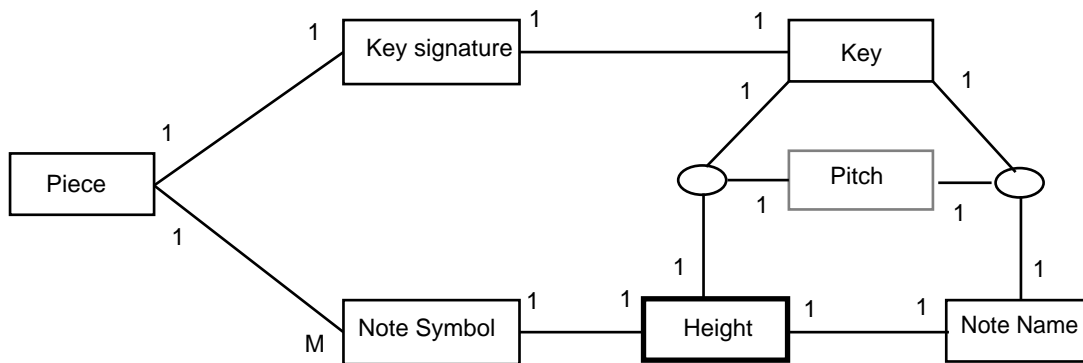
These benchmarks, though admittedly somewhat on the crude side, were applied by the authors to different languages and helped them find overlooked usability problems and encouraged them to redesign the languages. They encourage others to find similar domain-specific benchmarks for their own purposes.

Formalisation

The vagueness of the cognitive dimensions has stimulated more than one attempt to define satisfactory formal representations.

ERMIA

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:



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'	<i>change key signature</i>
foreach Note:	<i>change all notes</i>
read Note	
compute Note'	
write Note'	
<i>Total actions: #(write actions) = #(Notes)</i>	
<i>Working memory storage:</i>	
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!

Formalisation through System Modelling

Roast and Siddiqi (1996 et al.) take a very different 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 (1998) 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.

Part 2: Real Systems

In this part we shall work the other way round: instead of listing the dimensions and mentioning examples, we shall start with genuine applications and apply the cognitive dimensions approach to them.

It must be understood that doing a proper analysis, even in a broad-brush style, requires close familiarity with the device or notation. Since that degree of familiarity can hardly be achieved merely by reading about the device, the analyses given here will be rather sketchy.

The examples illustrate very different systems – an on-line diary and a pair of visual programming languages, showing how to go about applying the framework in these different cases.

A Diary System

On-line diaries and calendars, meeting organisers, time managers, and so on are one of the popular applications of information devices. The example we shall analyse is ‘Now Up-to-Date’, a product of Now Software, henceforth abbreviated NUD.

NUD is a complex calendar application designed to have lots of features yet be easy to use. Before applying individual dimensions we will outline its operation.

Events are the main type of data. Events can be dated (holiday, appointment); undated (e.g. to-do’s); and things like banners and graphics and notes, which are undated – so one might hesitate to call them events, but that’s what the device calls them.

Attributes of events include

- name
- type
- category (see below)
- priority
- reminder (has a default)
- start time
- duration
- whether to carry forward

Not all attributes apply in every case – e.g. a note doesn’t have a start time. Events may also be *repeating* – see below.

Categories describe events. Every event is in one and only one category: users can create new categories ad lib. For instance, a business category would contain work-related appointments and to-dos; there might be a holidays category, etc. One category must always be the default category; new events with unspecified category are defaulted to that category.

Sets determine which categories are visible; only one set is visible at once. They can contain any number of categories and can overlap as much as desired. (E.g. you might have a work set containing the business and holidays category, and a leisure set containing the holidays category plus the category of meetings of hobby clubs, and another set for your spouse/partner using the same system ...)

Several different *views* are available, some by time unit (year, month, day) and some as lists of events. The most relevant ones are the month view, the day view, and the list view.

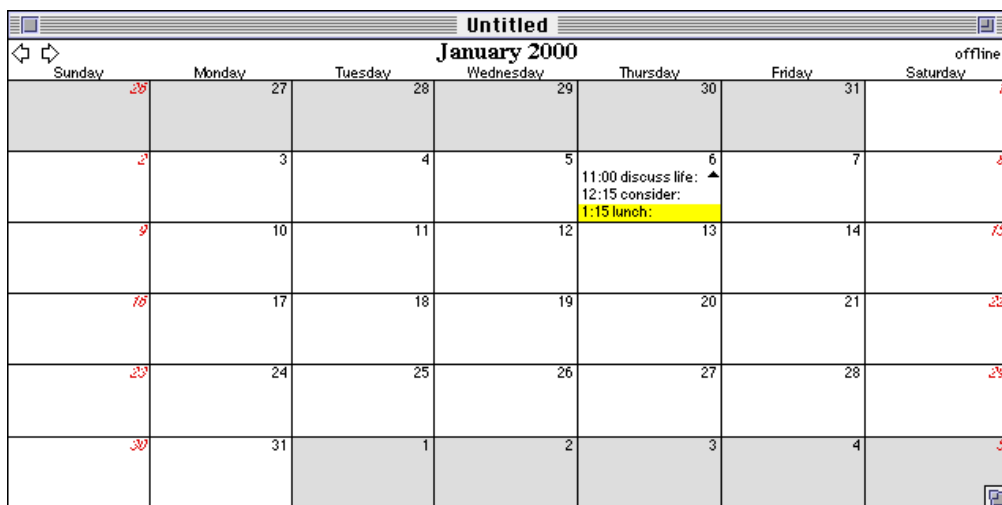


Figure 21 Month view in NUD.

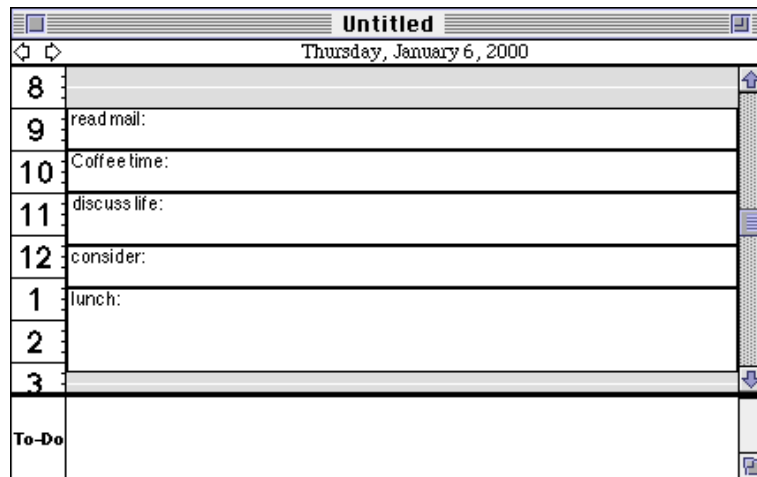


Figure 22 The day view in NUD.

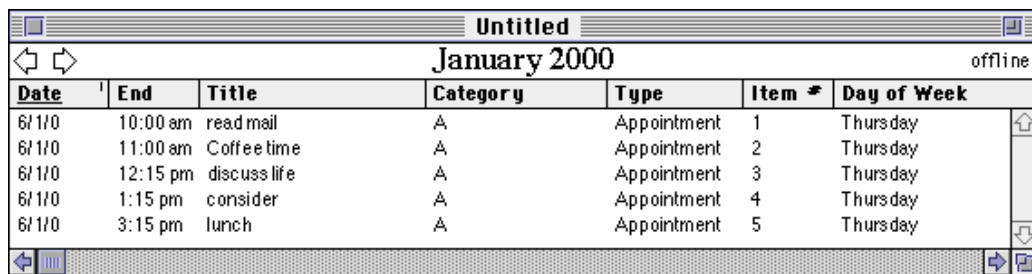


Figure 23 The list view in NUD. The user has considerable control over what categories are included in a list.

Cognitive dimensions: Main device

Viscosity

While in the month view, Modifying attributes (start time, etc) of the events that have been set up is very easy but requires typing in the time. Moving events from day to day is just drag-and-drop.

Inserting a new event in the month view potentially causes both knock-on and repetition. There is no way to insert an event between existing events and have them all close up slightly to make space for the newcomer, and so the user has to adjust all the existing events manually (this is the knock-on viscosity). In the month view, each event has to be separately opened and have its start and end time retyped (this is the repetition).

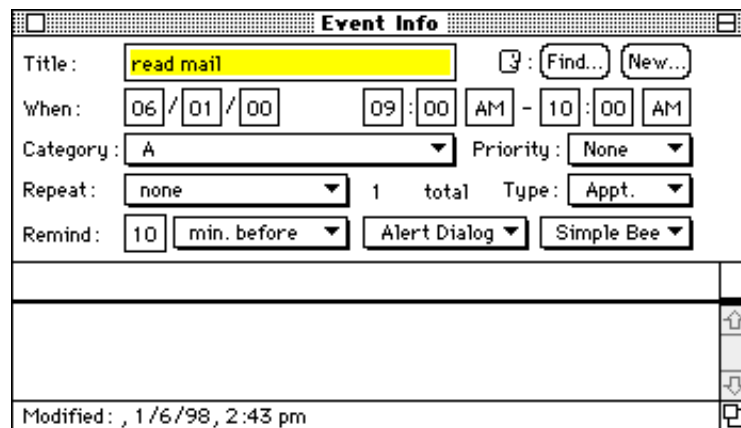


Figure 24 Defining a new event in NUD. To change the scheduled times of an existing event, this dialog box is re-opened and edited.

Fortunately, the day view is less demanding. Times are mapped onto spatial layout in a simple table, in which the start and finish times can be dragged up and down with the cursor:

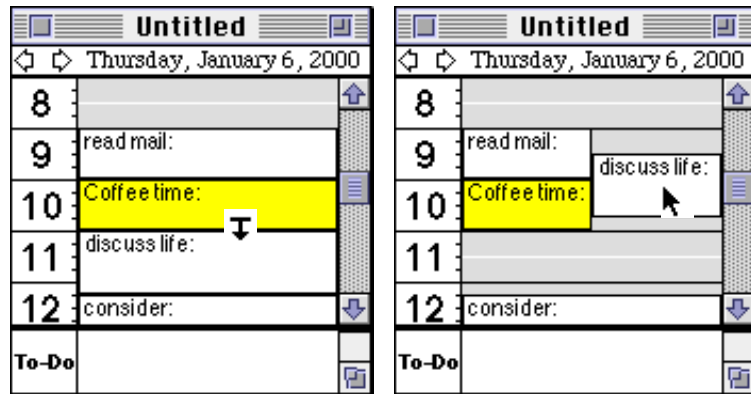


Figure 25 Left, modifying a day's schedule: coffee time is being extended. Right, rearranging the schedule by dragging 'discuss life' upwards. Note that overlapping events are shown side by side in the day view.

Although there is a small degree of repetition viscosity when inserting a new event into a busy schedule, the alternatives would probably have been too complicated.

Viscosity does, nevertheless, cause problems when a number of days are to be updated in the same way. If a set of appointments is to be carried over from day to day all week, or if something means that all appointments on each day for the next few weeks have to be put back half an hour, it will be a tedious job to enter all the changes. There is no facility, for instance, for copying a complete day and pasting it into the following day.

Hidden dependencies

There are very few dependencies in this system. Each event is independently recorded, and events can be allowed to overlap; thus changing one event will have no effect on others. But it should be noted that there is also no representation of dependencies that may exist in the user's life. One of the examples given by Payne (1993) is that of organising a conference, an activity spawning lots of sub-activities; if the conference is cancelled, all the associated dates and appointments can also be cancelled.

The system does, however, give a representation for events that repeat (birthdays, weekly seminars, etc), and there are some potential dependencies for these events, since different occurrences are necessarily linked. If the start or finish time of an individual event is to be altered, NUD offers a dialog box allowing the user to change all the events or only the currently selected event. But if an event is changed from a type with a start time to one that has no fixed time (such as a Note), then all the other repetitions of that event are silently changed, and their start times are lost.

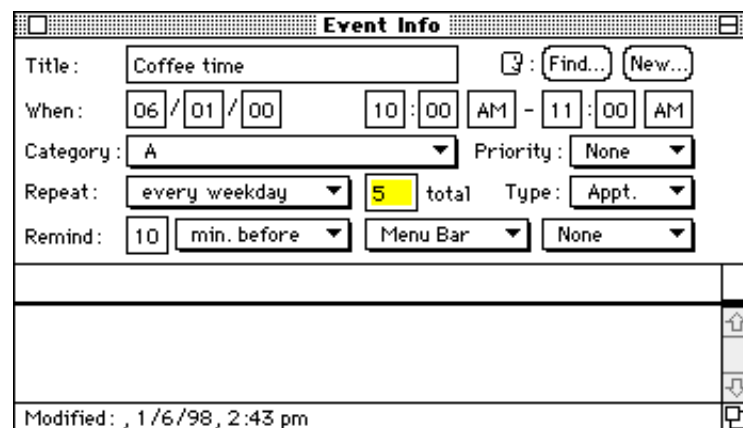


Figure 26 Defining a repeating event in NUD.

Premature commitment

In the main display, there is a modicum of premature commitment when defining the type of an event, since it must be classed as one of the set of types at the moment of creation; however, its type is very easily changed, so this is unimportant.

Abstraction Barrier and Abstraction Hunger

The abstractions available are categories and sets, described above, and repetitions of events. Every event belongs to just one category; sets may comprise any number of categories. Only one set is displayed at one time. The definition and editing of categories and sets is through an abstraction management sub-device, dealt with separately below.

The defaults allow the user to work in ignorance of category and set concepts until such time as they are needed – so there is no initial abstraction barrier. Thus, NUD is an abstraction-*tolerant* device.

Secondary Notation

Secondary notation is available through variations in colour, font and style. One use is to distinguish classes of event. Each set includes any number of categories, and categories may belong to any number of sets; and each set can have its own way to display each category. Thus a set called ‘Holiday events’ could display all the events put into the ‘Lolling around’ category in blue, and all the events in the ‘Seeing the sights’ category in red; while another set might include only one of those categories, and display it in green.

Any individual event can also be assigned its own over-riding display attributes, so the user could choose to show important events in big bold red letters if desired, regardless of the default attributes. Showing provisional events or cancellations is not so easy – they have to be treated as pseudo-events.

There is no escape from the formalism.

Visibility and Juxtaposability

Visibility for single events and classes is well handled. The day and month views illustrated above deal with most events; a list view can also be obtained, in which all events of a given set are listed together without the visual framework of the calendar:

Date	End	Title	Category	Type	Item #	Day of Week
6/1/0		discuss life	A	Special	1	Thursday
6/1/0	10:00 a	read mail	A	Appointment	2	Thursday
6/1/0	11:00 am	Coffee time	A	Appointment	3	Thursday
6/1/0	1:15 pm	consider	A	Appointment	4	Thursday
6/1/0	3:15 pm	lunch	A	Appointment	5	Thursday
7/1/0	11:00 am	Coffee time	A	Appointment	6	Friday
10/1/0	11:00 am	Coffee time	A	Appointment	7	Monday
11/1/0	11:00 am	Coffee time	A	Appointment	8	Tuesday
12/1/0	11:00 am	Coffee time	A	Appointment	9	Wednesday

Figure 27 The list view in NUD allows ‘timelines’ to be separated by displaying only the events of one category.

Juxtaposing days, weeks or months is possible by opening a duplicate window. This expedient is not immediately apparent and for a long time this user believed that there was no way to juxtapose weeks.

Abstraction management sub-device

The chief abstractions available are the repetition and the set of categories. Not much can be said about repetitions, but building and modifying sets is more interesting. The main display for constructing sets

of categories looks like the figure below. The list shows which categories are included in the set and what text attributes to use when displaying those events.

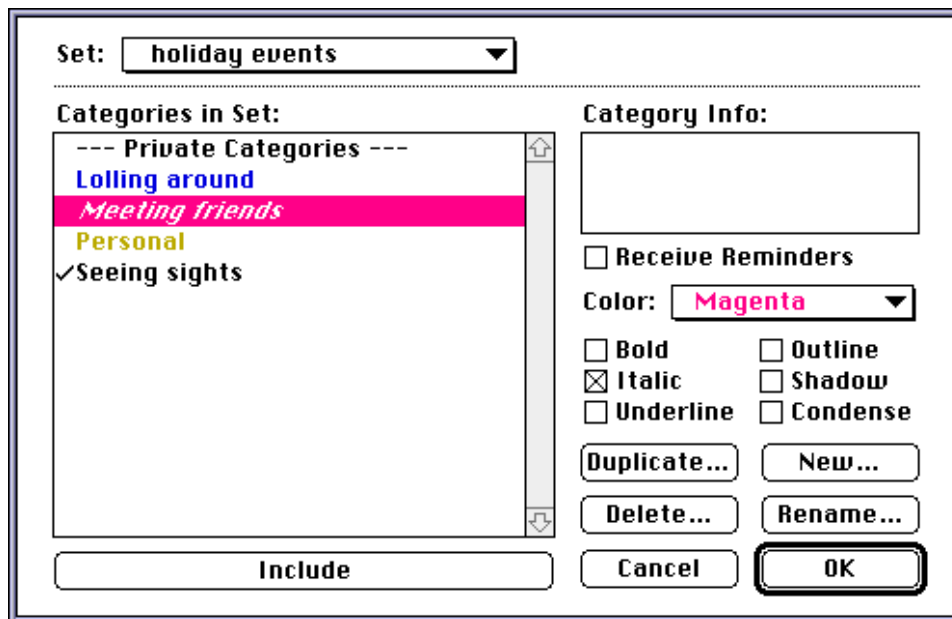


Figure 28 The dialog for managing sets of categories of events in NUD.

Viscosity

Viscosity is low: categories are readily introduced or removed.

Hidden dependencies

Few dependencies are involved.

Premature commitment

Building a set necessitates a certain amount of premature commitment for most users, since they do not know in advance whether their categorisation of their life events is stable or accurate. Fortunately, editing the set is not difficult.

Abstraction Barrier and Abstraction Hunger

NUD cannot be used without at least one category, to use as a default, and at least one set, again to use as a default. Although new users do not have to operate on these defaults, they cannot ignore them completely. There is, therefore, a real, though not enormous, abstraction barrier. The problems of new users would have been easier if better names had been found, indicating more clearly the relationship that every event \in exactly one category and every category \in at least one set.

Secondary Notation

There is no secondary notation for sets and categories.

Visibility and Juxtaposability

Visibility is good for single set definitions, as shown above, but there is no juxtaposition possible; only one set definition can be displayed at one time.

Conclusions

NUD is a good system for incrementation or transcription activity. Modification of existing schedules is less good.

Two Visual Programming Languages

Information-based devices are getting more sophisticated. At the rate we're going, our central heating controls will soon have macro languages and our video systems will have compilers and link loading. Maybe that's a slight exaggeration, but at the very least, we need to give increasingly complex instructions to devices like, for example, telephone-based query systems. How are users going to communicate what they want to happen?

Visual languages have been a frequent suggestion. Often the reasons given are based on rather suspect folk psychology, such as "graphical languages must be better because they use more of the brain" or that tired, tired slogan "A picture is worth ..." (see Blackwell, 1996, for a report on the folk psychology and its relationship to research findings).

But folk psychology aside, graphical languages have some degree of promise. They don't look like programming, which is a help: no grey slabs of small print, no tricky syntactic niceties. They are in disguise, the acceptable face of programming.

'Box-and-wire' diagrams are the commonest suggestion, though not necessarily the best: each box contains an instruction, and the wires either carry data from one box to the next (the data-flow paradigm) or indicate the sequence of operations, as in a flow-chart. We have seen box-and-wire languages proposed or implemented for many types of system, including

- scheduling systems
- network configuration systems
- component assembly environments
- Java generators
- project management tools
- telephone menu programming systems

How usable are 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 LabView2, and how they compared to the old-fashioned Basic language. This section is based on that work. (Only the highlights can be mentioned here; read the full paper before implementing your own box-and-wire system.)

Important note: both these languages are excellent for their purposes. In no way are the following remarks to be taken as a complete review. The two languages have adopted different solutions to the design issues and the trade-offs, and our purpose is merely to illustrate some of the consequences of exchanging one kind of usability for another kind. The first author, TG, has made much use of both languages and can warmly recommend them for their appropriate types of use.

Illustrating the notations

We cannot explain the details of these languages here, but we can give an idea of the important differences.

For comparative purposes, here are three 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

² Produced and trademarked by Pictorius Inc. and National Instruments Inc., respectively.

programming languages. First, we show the Basic version, the LabView equivalent, and lastly the Prograph version.

```
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 29 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.

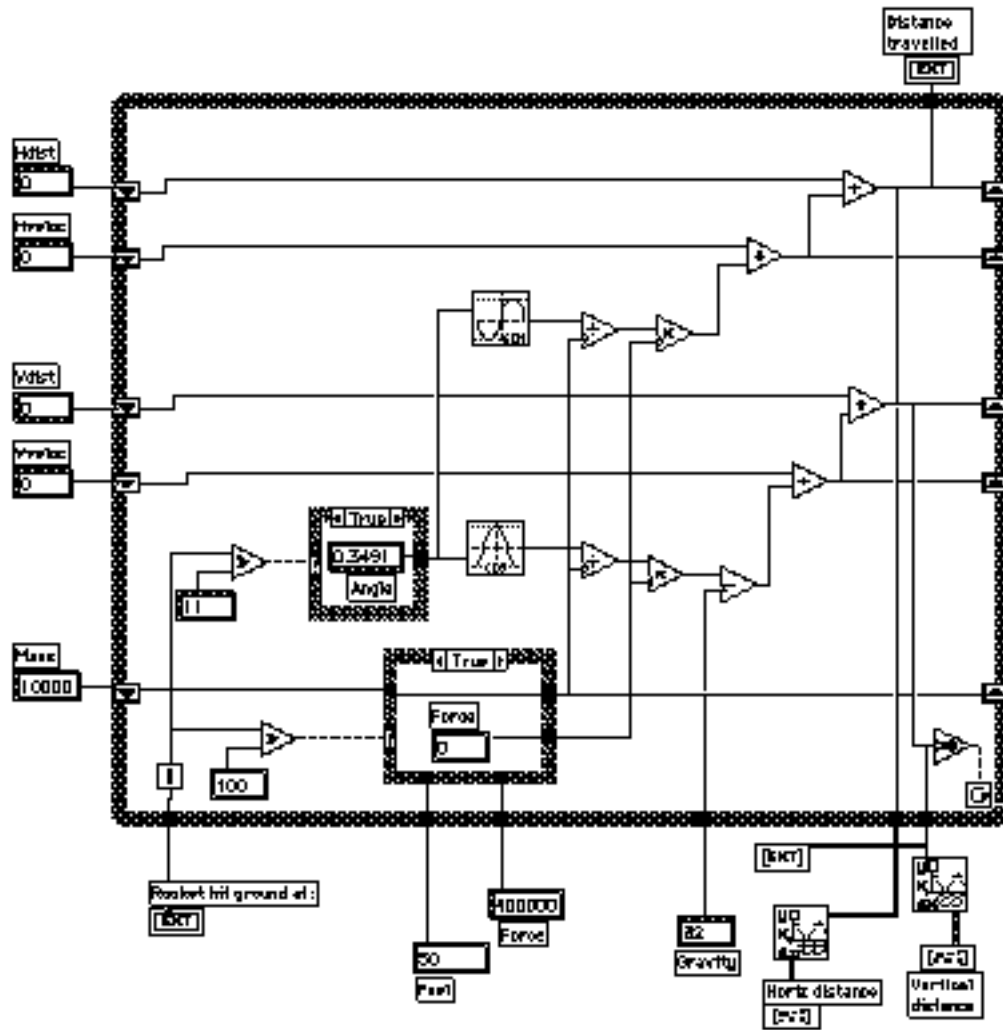


Figure 30 The rocket program in LabVIEW.

The Prograph notation likewise uses boxes to indicate operations and wires to transmit data, but puts each conditional into its own subroutine with its own window. The result looks like this:

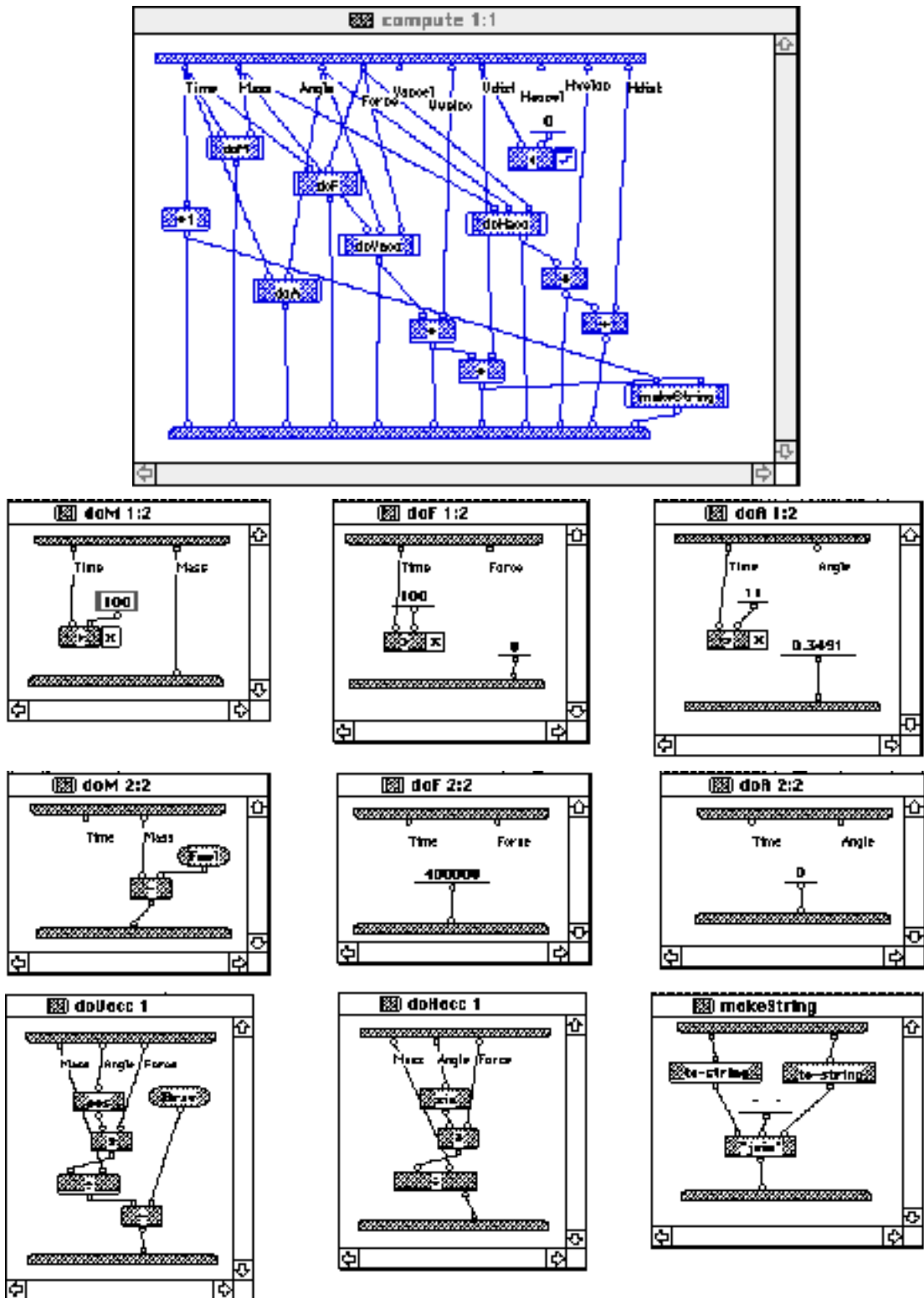


Figure 31 The rocket program in Prograph.

Viscosity

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. (Notice, however, that this causes poor visibility and juxtaposability, as mentioned below.) 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).

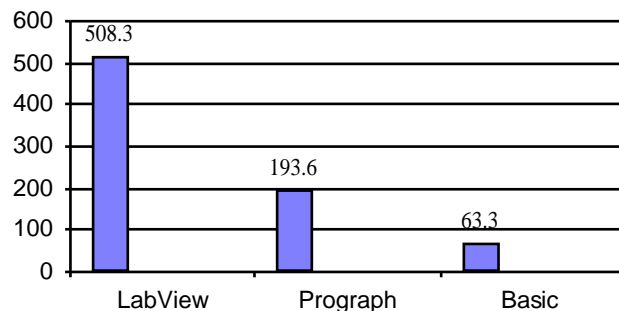


Figure 32 Times in seconds to make equivalent editing modifications

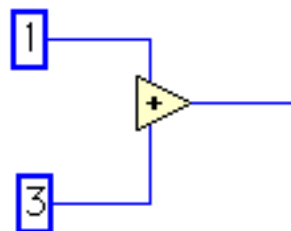
Viscosity is obviously a major issue.

Hidden dependencies

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
```

The same information in a LabView program is shown thus:



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.

Premature commitment

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.

The difficulty is better illustrated than described. Consider building a program to solve quadratic equations, according to the usual formula

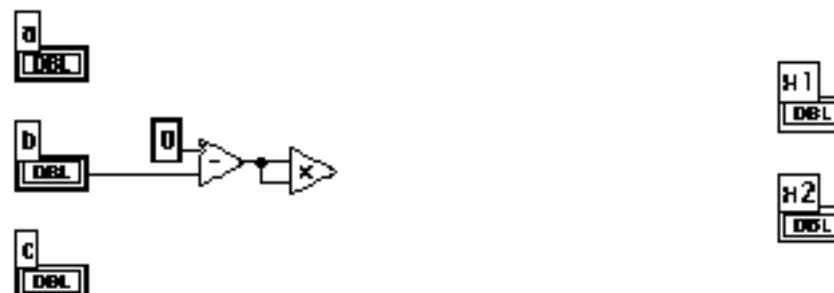
$$x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$$

The sequence of screenshots below I taken from an actual session in which a LabView user built the program.

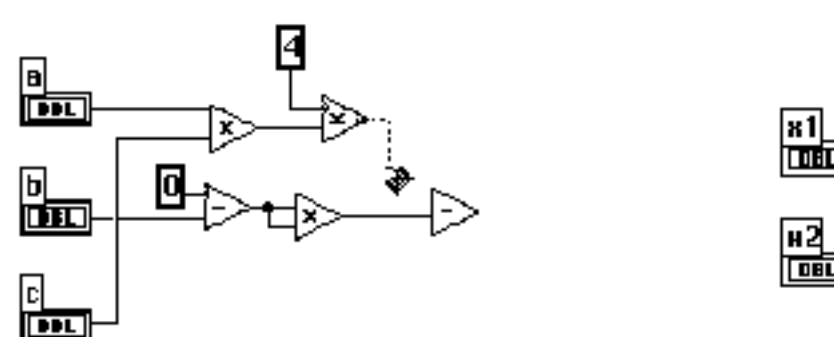
1. " ... Start with minus b..." :



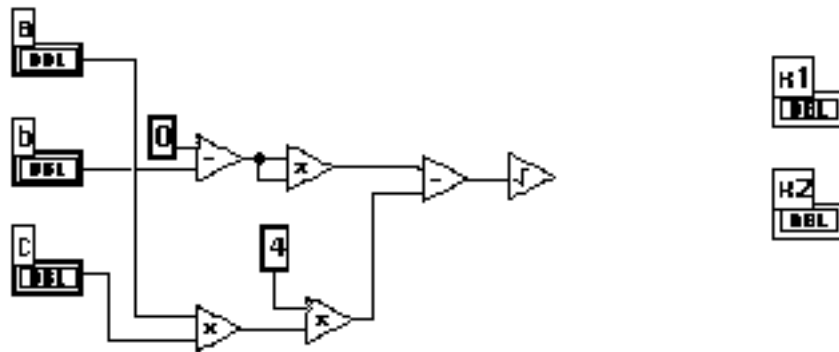
2. " ... I'll need b-squared too ..." :



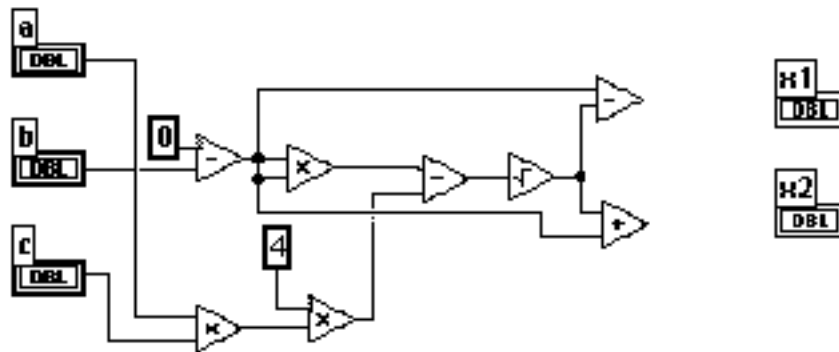
3. " ... Turn that into b-squared minus 4ac ..." : (Note: the wire bobbin in the next frame is the cursor of the connection-making tool.)



4. " --- Oops, that's going to be $4ac$ minus b -squared ... try moving that $4ac$ chunk down and reconnecting to the 'minus' box ..." :



5. " ... OK, now I need to do plus-or-minus that lot ..." :



6. "That's $\sqrt{b^2 - 4ac}$. But hey, What A Mess! – and I still haven't used the $-b$, let alone done the bottom line of the formula... How do I tidy it up?"

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. For example, in Prograph, the order of terminals is usually chosen to minimise wire crossings, but the lookahead to get this right is quite extensive, so even in the simple rocket program some unnecessary crossings occur. The next figure shows the 'visual spaghetti' created by not looking ahead – this example is drawn from practically the first method window opened in the first author's file of Prograph programs. The point here is not that it is possible to make a horrible-looking program; that's true in any language. The point is that it needs a lot of lookahead to avoid making a mess. Worse, because of the high viscosity of these languages, it takes too much time to clear it up, so the mess gets left.

Whitley and Blackwell (submitted) 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.

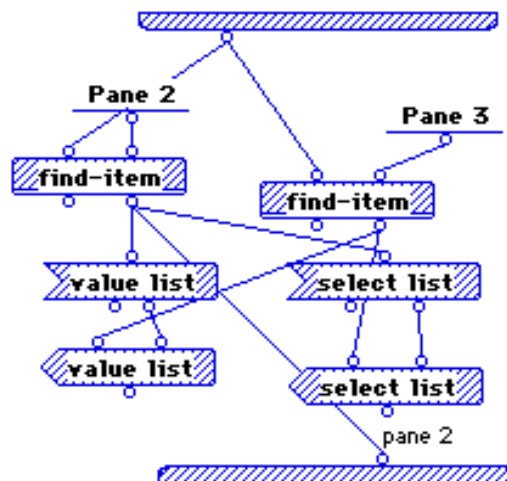


Figure 33 'Visual spaghetti'. To avoid this, the programmer has to look ahead. This example is in Prograph, but the same problem occurs in LabView or any other box-and-wire language.

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 Barrier and Abstraction Hunger

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.

Both languages have facilities for creating new operational abstractions. 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. Prograph, being object-oriented, requires some new abstractions to be created for most purposes, making it abstraction-hungry.

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

We shall not discuss the two different abstraction managers here.

Secondary Notation

Judging by these visual programming languages, designers have not thought hard enough about the need to escape from the formalism. Neither language has good facilities for commenting, for example. They each have the ability to attach a comment 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

```

Looking back to the LabView and Prograph programs, how would those comments be adapted?

Nor can these visual languages 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.

In short, secondary notation is a commodity in short supply in this particular design of visual language. The notation itself uses up the available parameters, leaving nothing over for expressing anything else. Bertin (1965) emphasises that in graphical notations the number of available variations is always limited, and they must be allocated wisely: in these notations, it seems, too many of the variations have been allocated to the notation itself.

Visibility and Juxtaposability

The visibility of data flow in the LabView language is excellent. Prograph dataflow is much less visible, the large number of small windows obscuring the flow and hindering search for specific items. The effect of the design decision to use separate windows in Prograph becomes all too apparent when one looks at the proliferation of windows.

Where LabView in turn 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 34 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.

References (Pts 1 and 2)

(For references not found here, such as Roast (1998), see the Further Reading section.)

Bertin, J. (1981). *Graphics and Graphic Information Processing*. Berlin: Walter de Gruyter.

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

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

Cramer, M. (1990) Structure and mnemonics in computer and command languages. *International Journal of Man Machine Studies*, 32 (6), 707-722.

Eisenstadt, M. (1993) Why HyperTalk debugging is more painful than it ought to be. In J. L. Alty, D. Diaper and S. Guest (Eds.) *People and Computers VII: Proc. HCI '93 Conference*. Cambridge: Cambridge University Press.

Fischer, G. (1988) Panel on Critical assessment of hypertext systems. *Proc. CHI 88 ACM Conf. on Human Factors in Computing Systems*, p 224. New York: ACM.

Neilsen, J. and Molich, R. (1990) Heuristic evaluation of user interfaces. In *Empowering people – CHI'90 conference proceedings*. ACM Press, New York.

Payne, S. J. (1993) Understanding calendar use. *Human-Computer Interaction*, 8, 83-100.

Payne, S. J. and Green, T. R. G. (1986) Task-Action Grammars: a model of the mental representation of task languages. *Human-Computer Interaction*, 2, 93-133.

Whitley, K. N. and Blackwell, A. F. (submitted for publication). Visual programming in the wild: a survey of LabVIEW programmers. Available on request as Technical Report CS-98-03, Computer Science Department, Vanderbilt University, Nashville, TN 37235, USA.

Part 3: Interactive Practice Examples

This section is devoted to practice exercises in the use of cognitive dimensions.

The exercises are intended for group discussion. Toy interactive applications have been created in pairs to illustrate contrasting design choices; the group is invited determine what trade-offs in the dimensions have been caused by the design difference. Thus there are two telephone systems, each with its own advantages; two central heating controls, each with its own advantages, and so on. Toy *non-interactive* applications – notations for displaying information – may also be included at a later stage.

The cognitive dimensions framework cannot be successfully applied until one is familiar with the device or notation being studied. For this exercise, it would be best to be familiar with both of the devices in a pair, to make it easier to see how advantages in one trade-off against advantages in the other. The group should therefore start by exploring each of the two versions and devising tasks that highlight their differences, looking especially for tasks that find an advantage for one of them. They should then identify the major design difference, apply the cognitive dimensions framework, and decide whether the trade-offs discovered fit the patterns described in earlier parts of this tutorial. A final and important step is to think of real-life examples of each alternative, preferably from commercial applications, and to decide whether they support the analysis made by the group.

Toy applications (widgets) have been used for several reasons:

- a single design choice can be isolated and its consequences compared in alternative versions;
- they are small enough to be delivered over the web, and will continue to be available for use on other occasions – for example, if attendees wish to pass on to others what have learned, these examples can be re-used;
- exploring and analysing a full-scale application would take too long for class use.

Web delivery

The widget examples are accessible via the URL

<http://www.ndirect.co.uk/~thomas.green/devices/>

Most of them are written using JavaScript, and these ones work with Netscape Navigator 3 on Macs, PCs, and Suns. Make sure that JavaScript has been enabled (open Options -> Languages and check the JavaScript box).

There is no guarantee that the JavaScript widgets work with any version of Internet Explorer or any other browser.

One widget pair has been written in Java. At the time of writing there is some technical difficulty in making this widget work with any browser except Sun's applet viewer. Let us hope that is resolved before you read this

Aims

The aims are:

- to identify differences in their cognitive dimensions;
- and to observe how the design choice has traded off some dimensions against others.

As a result of this exercise, you may decide that one widget is better than another for certain types of activity, while the other is better for different types of activity.

What to do

Find the widgets;

Become familiar with each;

Locate any sub-devices;

Devise tasks that highlight their differences, looking especially for tasks that illustrate competing advantages;

Note the effects of the medium (persistent or transient?) and consider the most likely types of activity that users will engage in (exploratory creation / transcription / modification / incrementation);

Apply the cognitive dimensions framework to each of them and to their sub-devices, if present (this may prompt you to think of further tasks);

Note what the trade-off relationships are;

Try to think of further real-life examples of each version.

Remember!

The aim of this exercise is not to decide that one widget is 'better' than the other. It is very unusual for one design to be better than another for every conceivable purpose, *so don't be simple-minded*. Don't just say 'this one is obviously better' (are you sure it's better for *everything?*): instead, look for the cognitive dimensions trade-offs.

The trade-offs may be easy to spot in these small examples, but they will be less obvious in a larger example, so practice is beneficial.

Find Generalisations

An important part of the exercise is to think of further real-life examples of the same principles.

The Widgets

Form-Filling and Menu Choices

Users often have to make a set of choices that are inter-dependent, such that each choice influences the acceptable options for all other choices. A notably repulsive case is answering questions over the telephone while a clerk fills out a form; too late you realise that you should have said No at an earlier stage...

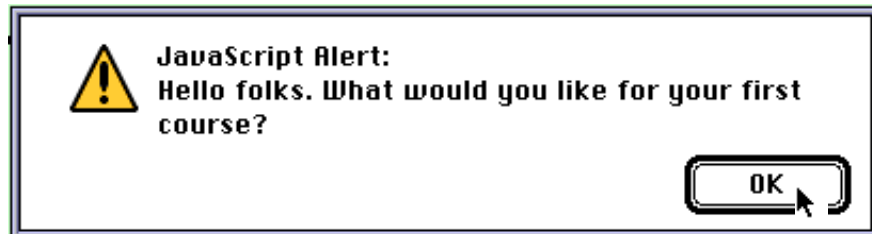
That abstract scenario has been realised in these widgets as choosing courses for a meal. You have to make your choices under the following constraint:

No two courses should contain the same ingredient. E.g. you should only have fish in one course. Each vegetable counts as a unique ingredient, so if you have asparagus in one course you must avoid it in subsequent courses.

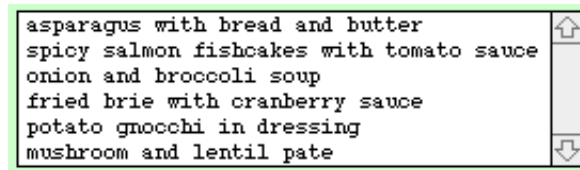
At Disobliging Dave's, the waiter gives you the choices for each course independently. At Cheerful Charlie's, you fill out a form for the whole meal at one go.

Disobliging Dave's

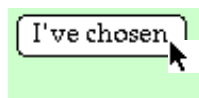
Snap your fingers to summon the waiter. He, she or it says:



When you press OK, the waiter offers you a selection:

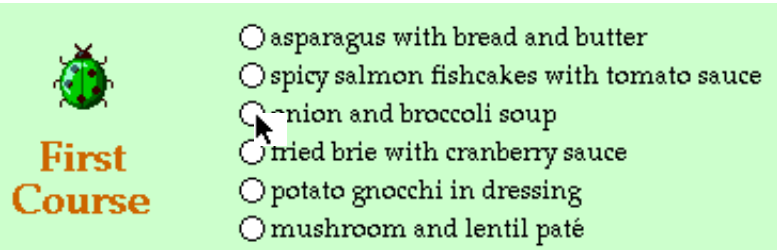


Click on your choice, then press the "I've chosen" button.



Cheerful Charlie's:

The whole menu is on one form. Each course is a set of alternatives:



Just set up the form as you want it – that's all you have to do.

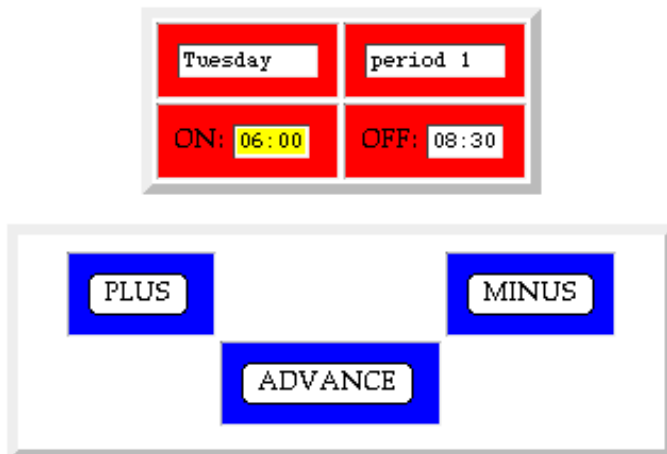
Controlling the Heating Controller

Once a set of choices has been made for one occasion, users may have to make the same choices for several other occasions: and not infrequently, for some of those occasions, the choices will be identical or only slightly different, so the user has to play the same moves again.

That abstract scenario has been realised as setting the domestic heating controls for each day of the week. The heating system to be set has a clock to control three heating periods, each with its own ON and OFF times, so you can set the heating to be (say) on from 7 am to 8 am, then again from 6 pm to 8 pm, and once more from 10 pm to 11 pm. (If the on and off times for a period are the same, the heater does not switch on for that period.)

One controller, the Balmoral, just sets each of the times. The other controller, the Alhambra (based on a real system, the Horstmann Channel Plus), has an added facility, a Copy function. If (say) Tuesday is selected, then the effect of Copy is to make all Tuesday's on-off times identical to Monday's times and then to select the next day, Wednesday. Pressing Copy 6 times sets all the on-off times to be the same, right through the week.

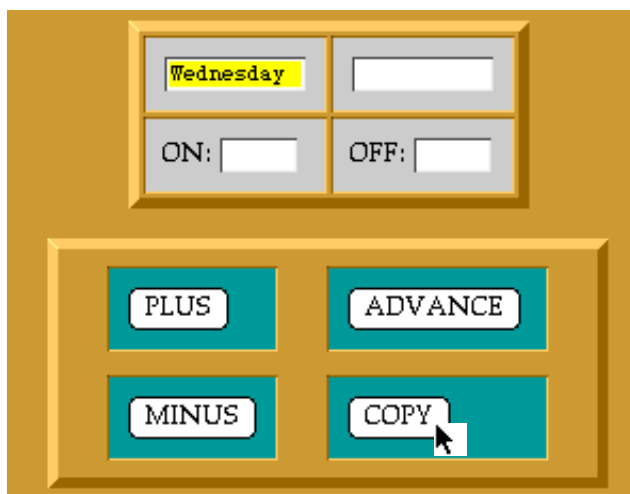
The Balmoral Model



In this model, Plus and Minus control the setting of the selected item (in the illustration here it's the 06:00, shown shaded; it's highlighted on the screen).

The Advance button selects Period -1-On, Period-1-Off, ditto for Periods 2 and 3, then selects next day: at that point, the day is shown highlighted with no times visible (as shown in the diagram of the Alhambra system). Pressing Plus will then move immediately to the following day.

The Alhambra Model:



As for Balmoral, except that while the day is selected (as shown here), pressing the Copy button will set all that day's on-off times to those of the preceding day and will then select the next day.

Number Games with Telephones

Telephones and telephone numbers have been mentioned several times in this tutorial. Here is one to analyse, based on the British Telecom Relate 80 handset, which has a memory that can store up to ten numbers.

There is only one version of this widget – just compare its functioning with and without the memory.

Making a call



- ‘Lift the phone’ by pressing the top left radio button;
- Then press the number keys in the appropriate order.
- That’s all you have to do.
- ‘Hang up’ by pressing the top right-hand radio button.

Using the memory

(The following are essentially the manufacturer’s instructions.)

To store a number:

- pick up the handset
- press STORE
- key in the telephone number you wish to store
- press STORE again
- press one of the keypad buttons (0 - 9) (depending on where you wish to store it)
- the number is now stored

To dial a stored number:

- pick up the handset
- press RECALL
- press one of the keypad buttons for the number you want to dial

The Memory Label

1	HCI: 0114-225-5335
2	
3	
4	
5	
6	
7	
8	
9	
0	

**MEMORY LABEL -
note your settings
here**

The BT Relate 80 comes with a holder for a piece of card on which you can note the stored numbers. The manufacturer's instructions say:

- Write the name of the person whose number is stored on the memory label.
- (Use a pencil, so it can be changed if necessary.)

(How does the recommendation to use a pencil affect the cognitive dimensions?)

For the interactive widget, of course, you should type in the stored numbers.

Tiling in Style

The telephone example illustrated one form of re-use. Here is another, from a different domain in which the medium has different characteristics, and the probable user activities are different.

This is a an editor for use when defining tile floors. In one scenario we have envisaged, the editor is to be used by a Tile Consultant. A customer will supply a sketch of the desired design or maybe give an verbal informal specification, and the Tile Consultant will define a suitable design.

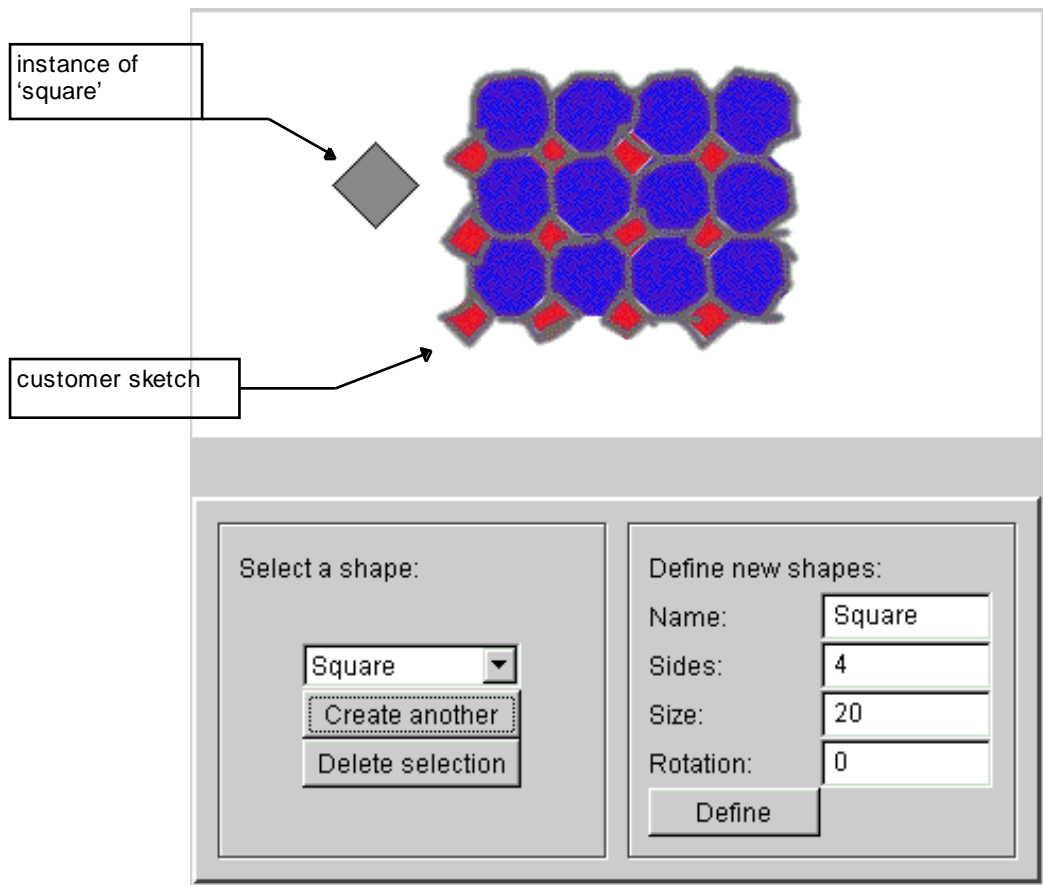
In an alternative scenario, the customer is a direct end-user of the tile editor, exploring possibilities without any form of preliminary sketch.

The Greenwich model

In this model, the user defines any number of tile shapes, then arranges these shapes to match a sketch that has been supplied by the customer.

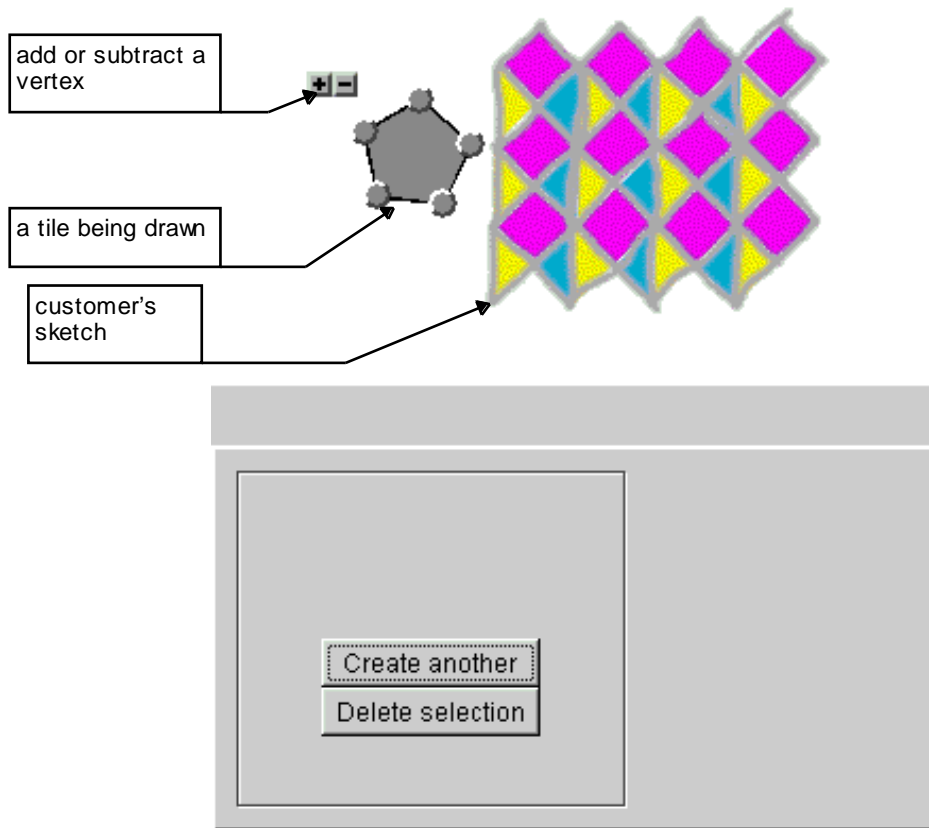
Each shape of tile must be defined as a new abstraction before it can be used in the design; thereafter, that shape can be freely re-used.

Redefining a shape changes all the instances that have already been used.



The Harrogate Model

An alternative version of the editor allows tiles to be defined using direct manipulation, so that no abstraction needs to be created. Each tile is drawn (and modified) individually.



Further Reading

Following is a partial list of publications wholly or partly related to research on cognitive dimensions.

Buckingham Shum, S. and Hammond, N. (1994) Argumentation-based design rationale: what use at what cost? *International Journal of Human-Computer Studies* 40 (4), 603-652.
<http://kmi.open.ac.uk/~simonb/DR.html>

Includes a cognitive dimensions analysis of visual design rationale tools

Gilmore, D. J. (1991) Visibility: a dimensional analysis. In D. Diaper and N. V. Hammond (Eds.) *People and Computers VI*. Cambridge University Press.

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.

The original paper

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.) *Human-Computer Interaction – INTERACT '90*. Elsevier.

Green, T. R. G. (1991) Describing information artefacts 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., Winder, R., Gilmore, D. J., Davies, S. P. and Hendry, D. (1992) Designing a cognitive browser for object-oriented programming. *Artificial Intelligence and Simulation of Behaviour Quarterly*, Issue 81, 17-20.

Green, T. R. G. (1996) The visual vision and human cognition. Invited talk at Visual Languages '96, Boulder, Colorado. Abstract only appears in: *Proceedings of 1996 IEEE Symposium on Visual Languages*. Eds: W. Citrin and M. Burnett. Los Alamitos, CA: IEEE Society Press, 1996.

<http://www.ndirect.co.uk/~thomas.green/workStuff/VL96Talk/VLTalk.html>

Adapts much of the content of Green and Petre (1996) for web presentation

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

gzip (180K): <ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/thomas.green/VPEusability.ps.gz>

uncompressed (2.2 Mb): <ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/VPEusability.ps>

A lengthy analysis of selected visual programming languages: the most extensive presentation so far in print

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

gzip (80K): ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/Skull_v2.ps.gz

uncompressed (305K): ftp://ftp.mrc-apu.cam.ac.uk/pub/personal/tg/Skull_v2.ps

PDF (121K): <http://www.dcs.napier.ac.uk/~dbenyon/IJHCSpaper.pdf>

Goes a little way to formalising the dimensions in terms of extended entity-relationship models

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.

Includes a cognitive dimensions analysis of spreadsheet pros and cons

Hendry, D. G. and Green, T. R. G. (1993) CogMap: a visual description language for spreadsheets. *Journal of Visual Languages and Computing*, 4(1), 35-54 (special issue on cognitive aspects of visual languages)

How to improve spreadsheets, according to cognitive dimensions analysis (particularly with regard to 'role-expressiveness', not explored here)

Hendry, D. G. and Harper, D. J. (1997) An informal information-seeking environment. *Journal of the American Society for Information Science*, 48(11):1036-1048.

Considers information seeking as opportunistic design of queries, to which cognitive dimensions can be applied, and relates usability of systems to the ease of forming and modifying the queries.

Lavery, D., Cockton, G., and Atkinson, M. (1996). Cognitive Dimensions: Usability Evaluation Materials, Technical Report TR-1996-17, University of Glasgow.

<http://www.dcs.gla.ac.uk/~darryn/research/publications/TR-1996-17/>

Part of a study comparing effectiveness of different evaluation methods; need to contact the author for full details

Modugno, F. M., Green, T. R. G. and Myers, B. (1994) Visual programming in a visual domain: a case study of cognitive dimensions. In G. Cockton, S. W. Draper and G. R. S. Weir (Eds) *People and Computers IX: Proc. BCS HCI Conference*. Cambridge University Press.

Analyses a programming-by-demonstration environment, leading to a redesign

Petre, M. and Green, T. R. G. (1990) Where to draw the line with text: some claims by logic designers about graphics in notation. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) *Human-Computer Interaction – INTERACT '90*. Elsevier.

Petre, M. and Green, T. R. G. (1992) Requirements of graphical notations for professional users: electronics CAD systems as a case study. *Le Travail Humain*, 55(1), 47-70

Roast, C. R. and Siddiqi, J. I. (1996) Formally assessing software modifiability. In C. R. Roast and J. I. Siddiqi, editors. *BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, Sheffield Hallam University, 10-12 September 1996, Electronic Workshops in Computing. Springer-Verlag, 1996.

PS: <http://homepages.shu.ac.uk/%7Ecmscrr/PAPERS/fahciOnline.ps>

Roast, C. R. (1998) Modelling premature commitment in information artifacts.. To appear in *EHCI Proceedings* (ed: L. Nigay).

<http://homepages.shu.ac.uk/%7Ecmscrr/PAPERS/premcom.ps>

Roast, C. R. and M. B. Özcan. (1997) Cognitive dimensions applied to modifiability within an integrated prototyping environment. In R. Osborn and B. Khazaei, editors, *The Psychology of Programming*, PPIG97 Workshop Proceedings, pages 17-30. Computing Research Centre, Sheffield Hallam University,

PS: <http://homepages.shu.ac.uk/%7Ecmscrr/PAPERS/ppig97.ps>

Roast, C. R. and Siddiqi, J. I. (1996) The formal examination of cognitive dimensions. In A. Blandford and H. Thimbleby, editors, *HCI96 Industry Day & Adjunct Proceedings*, pages 150-156, 1996.

Roast, C. R. and Siddiqi, J. I. (1996) Relating knock-on viscosity to software modifiability. *Proceedings of OZCHI 96*, Hamilton, New Zealand, pages 222-227, 1996

PS: <http://homepages.shu.ac.uk/%7Ecmscrr/PAPERS/ozchi96.ps>

- Shum, S. (1991). Cognitive dimensions of design rationale. In D. Diaper and N. V. Hammond, (Eds.) *People and Computers VI: Proceedings of HCI'91*, 331-344. Cambridge: Cambridge University Press.
<http://kmi.open.ac.uk/~simonb/DR.html>
- Simos, M. & Blackwell, A. F. (1998). Pruning the tree of trees: The evaluation of notations for domain modelling. In J. Domingue & P. Mulholland (Eds.), *Proceedings of the 10th Annual Meeting of the Psychology of Programming Interest Group*, pp. 92-99.
- This discusses the more abstract requirements of notations used in the field of Domain Modelling, and considers how theories of metaphor in notation, as well as Green's Cognitive Dimensions, can be used to analyse them.
- Stacey, M. K. (1995) Distorting design: unevenness as a cognitive dimension of design tools. In G. Allen, J. Wilkinson & P. Wright (eds.), *Adjunct Proceedings of HCI'95*. Huddersfield: University of Huddersfield School of Computing and Mathematics, 1995.
<http://www.mk.dmu.ac.uk/~mstacey/Documents/uneven.htm>
- Proposes a new dimension
- Tweedie, L. (1995) Interactive visualisation artefacts: how can abstractions inform design? In Kirby, M. A. R., Dix, A. J. and Finlay, J. E. (Eds.), *People and Computers X, Proc. HCI'95 Conference*. Cambridge University Press.
- Proposes an approach derived from cognitive dimensions specialised for interactive visualisation tools
- Wilde, N. P. (1996) Using cognitive dimensions in the classroom as a discussion tool for visual language design. *Electronic Proceedings CHI 96*, Eds R. Bilger, S. Guest, M. Tauber.
http://www.acm.org/sigs/sigchi/chi96/proceedings/shortpap/Wilde/wn_txt.html
- How real are the cognitive dimensions? Do they help teaching?
- Wood, C. C. (1993) A cognitive dimensional analysis of idea sketches. Cognitive Science Research Paper 275, School of Cognitive and Computing Sciences, University of Sussex.
<ftp://ftp.cogs.susx.ac.uk/pub/reports/csrp/csrp275.ps.Z>
- The cognitive dimensions framework was used to interpret a study of collaborative idea sketching, giving a cohesive and comprehensive characterisation of the pertinent properties of idea sketches, as gathered from interviews and observation with 128 academics.
- 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.
- Describes simple but effective metrics for selected dimensions, leading to redesign of two different languages.

Index of examples

alarm clock, 36
benchmarks, 44
C, 27
CAD, 26
calendar, 31
calendars, 47
Cameras, 35
circuit diagram, 41
circuit diagrams, 40
circuit notations, 30
comment, 32
Fax machines, 35
Flowcharts, 27
folk psychology, 52
formalisms, 31
form-based system, 36
front panel, 30
global find-and-replace, 26
grouping, 26
HyperTalk, 27
in-car audio, 39
In-car radios, 35
macro, 26
meeting organisers, 47
menu structure, 35
menus, 39
object hierarchies, 26
Pencils, 41
Powerpoint templates, 27
Programming By Example, 28
scripting languages, 36
Smalltalk, 27
Spreadsheets, 27, 37, 40
Style sheets, 26
telephone memories, 26
telephone number, 30, 34
visual programming language, 39
Word styles, 26