

Many believe that visual programming techniques are quite close to developers. This article reports on some fascinating research focusing on understanding how textual and visual representations for software differ in effectiveness. Among other things, it is determined that the differences lie not so much in the textual-visual distinction as in the degree to which specific representations support the conventions experts expect.



Why Looking Isn't Always Seeing: *Readership Skills and Graphical Programming*

M A R I A N P E T R E

"A picture is worth a thousand words"—isn't it? And hence graphical representation is by its nature universally superior to text—isn't it? Why then isn't the anecdote itself expressed graphically? Perhaps anecdotes don't lend themselves to purely graphical presentation. Perhaps this phrase is too simplistic to be appropriate in the context of graphical notations. Nevertheless, many writers on visual programming argue in just this way: graphical representations are better simply because they are graphical (e.g., [22]).

This article argues otherwise: that text and graph-

ics are not necessarily an equivalent exchange, and that we still don't fully understand 'what's good about graphics'. This is not an argument for a 'textist opposition', but rather a call for balance and consideration. Both graphics and text have their uses—and their limitations. Pictorial and graphic media can carry considerable information in what may be a convenient and attractive form, but incorporating graphics into programming notations requires us to understand the precise contribution that graphical representations might make to the job at hand.

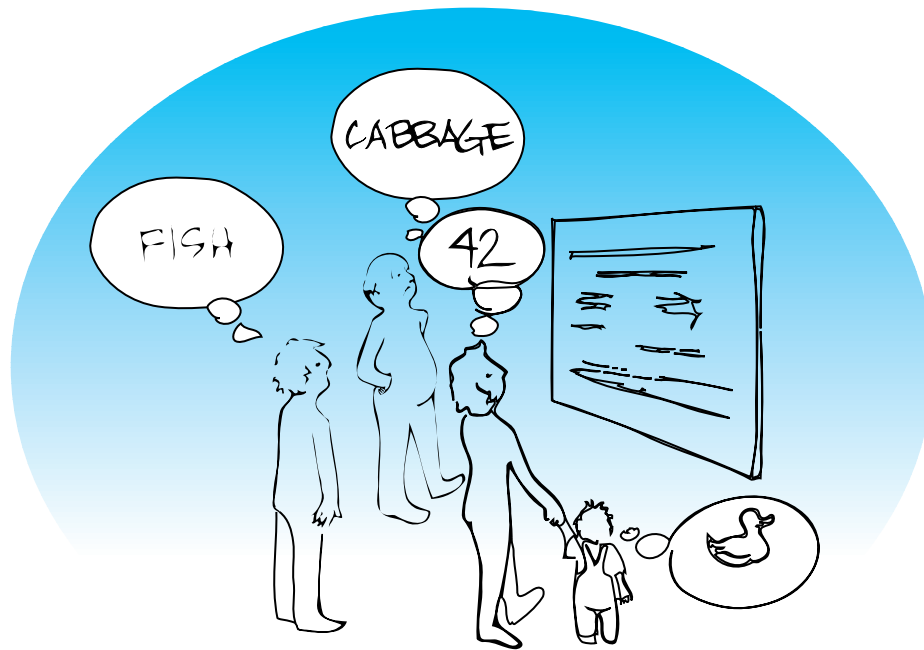


Figure 1.

In notation: does a given picture convey the same thousand words to all viewers?

tual ones is that the programmer takes in a program in the same way that a viewer takes in a painting: by standing in front of it and soaking it in, letting the eye wander from place to place, receiving a ‘gestalt’ impression of the whole. But one purpose of programs is to present information clearly and unambiguously. Effective use requires purposeful perusal, not the unfettered, wandering eye of the casual art viewer. The aim is not poetic interpretation, but reliable interpretation. The question is not ‘Is a picture worth a thousand words?’, but ‘Does a given picture convey the same thousand words to all viewers?’ (see Figure 1).

A programmer is more like the reader of a technical manual than the viewer of a painting: a deliberate reader, goal-directed and hypothesis-driven. Some studies of reading clearly show that accomplished readers, reading for comprehension, are deliberate readers, making great use of the typographic and semantic cues found in well-presented text (see [2]). To support them in this activity, typographers have evolved ways—graphical enhancements—to make required information quickly accessible (program comprehension is analyzed in this style in [16]).

The programmer uses a programming notation with specific tasks or goals in mind, tasks that may well be complex and heterogeneous. The success of a representation, graphical or textual, depends on whether it makes accessible the particular information the user needs—and on how well it copes with

the different information requirements of the user’s various tasks.

Graphical representations are more challenging than they appear at first. This article refers to research results to consider why the attractions of graphical representations are not matched by performance, putting forth the arguments:

- that much of what contributes to the comprehensibility of a graphical representation isn’t part of the formal programming notation but a ‘secondary notation’ of layout, typographic cues, and graphical enhancements that is subject to individual skill;
- that graphical readership is an acquired skill: structure, relationships, and relevance aren’t universally obvious;
- that experts ‘see’ differently and use different strategies from novice graphical programmers;
- that, although some of their touted qualities may be illusory, graphical representations are nevertheless persistently appealing and that this appeal may have its own value;
- that the role of graphics in notation must be addressed realistically, rather than simplistically.

This article discusses these observations about how programmers actually use different representations and challenges the naive assumption that graphical representations are unproblematically more ‘transparent’—more accessible, comprehensible, and memorable—than textual ones. It suggests instead that no single representation is a panacea, but that we need to identify appropriate criteria for choosing representational ‘horses’ for cognitive ‘courses’.

This argument focuses on the use of graphics in notation, specifically in graphical programming languages. The issues addressed generalize to other contexts, such as computer interfaces and environments, where information must be presented precisely.

Graphics Do Not Guarantee Clarity: 'Good' Graphics Relies on Secondary Notation

The strength of graphical representations—almost universally—is that they complement perceptually something also expressed symbolically. For instance, when functionally-related components are placed close together, which is typical practice in electronics schematics, an analog mapping is being used to supply extra information over and above the information explicitly represented by the components and their connections. Expert designers regard this 'secondary notation' as being crucial to comprehensibility.

We adopted the term 'secondary notation' to refer to such analog mappings and to emphasize that these valuable layout cues are typically not formally part of the notation—are 'secondary' to the language definition—but that they can be used to exhibit relationships and structures that might otherwise be less accessible. In observational studies [17, 18], we interviewed a number of expert digital electronics designers and observed them at work in order to investigate their use of graphics and text in electronics schematics—a well-evolved, largely graphical notation supported by sophisticated CAD systems. These studies highlighted the notion of 'secondary notation': the use of layout and perceptual cues (elements such as adjacency, clustering, white space, labelling, and so on) to clarify information (such as structure, function, or relationships) or to give hints to the reader.

Secondary notation may well be the principal characteristic that distinguishes graphical notations. Raymond [20] argues that the possibility of analog mapping is the only specifically visual contribution of graphical programming languages, and that other characteristics of contemporary graphical programming languages can be realized just as well in textual languages. 'Good' graphics usually means linking perceptual cues to important information.

This is the "Catch-22" of secondary notation: that a major determinant of 'good' graphics is not part of the formal system. The mere presence of graphical features does not guarantee clarity in a representation. What is required in addition is good use of secondary notation, which—like good design—is subject to personal style and individual skill. Poor use of secondary notation is one of the things that typically distinguishes a novice's representation from an expert's—a difference visible to the experts we interviewed, for example, *The difference in the design between mechanically identical designs done by a novice and an expert, is the novice engineer's will be more difficult to comprehend because of the layout.*

In the examples we gathered, novice drawings failed in typical ways to exploit the available manipulable cues; for example, they grouped things visually that were not related logically, they dispersed related components, they confused the signal flow, they neglected conventions. Worse, they introduced a number of mis-cues, using strong perceptual signals

like symmetry and grouping arbitrarily, so that the reader is led to expect associations or functions that are not present (see figure 2). Similarly, programmers often fail to take advantage of local organization that is available in text (see figure 3). It is important to recognize that poor use of secondary notation isn't merely neutral, it can confuse and mislead.

Experience and personal skill play an important role in the exploitation of secondary notation in the production of graphical representations. Subsequent experimental studies show that they play a role in comprehension as well.

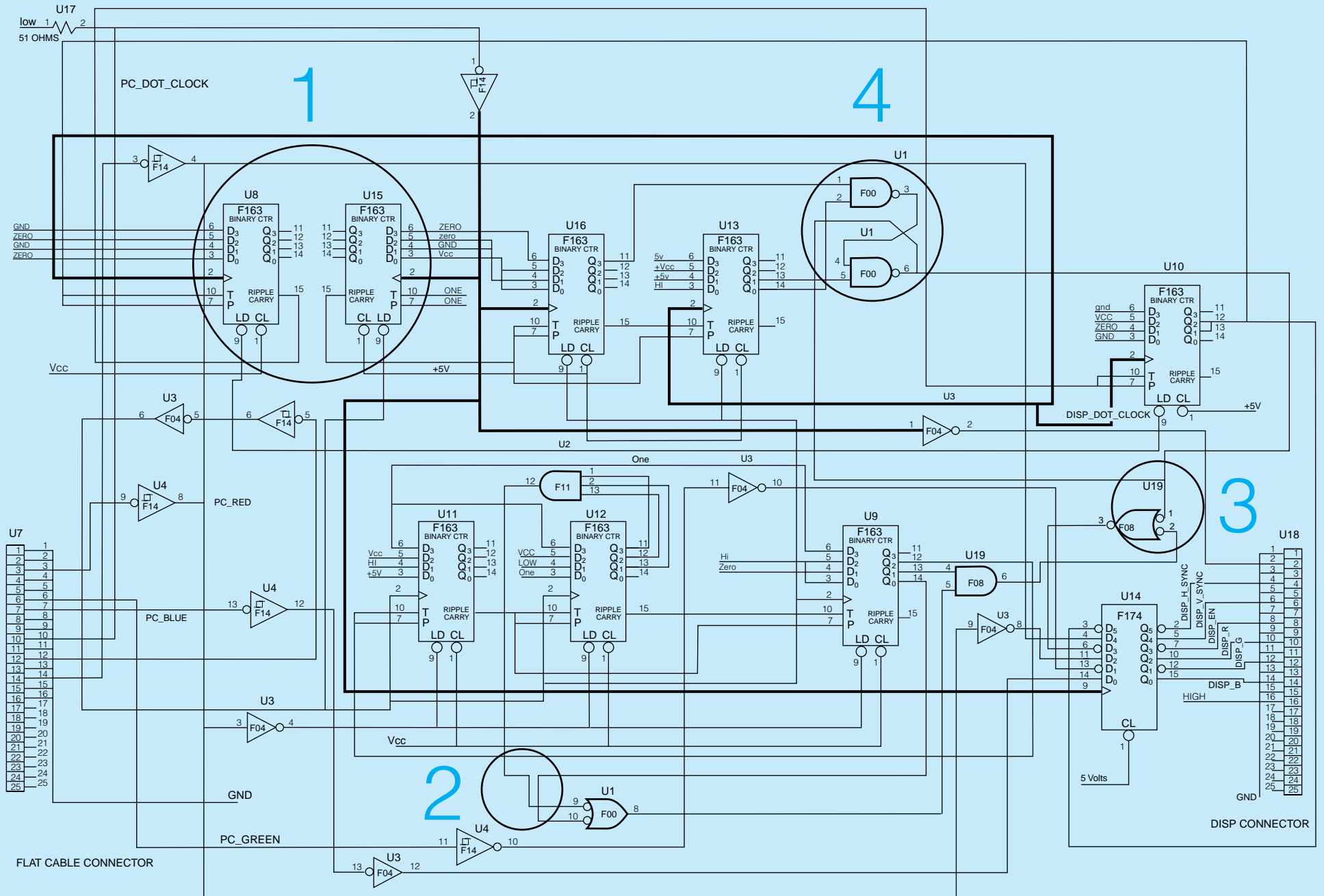
Knowing Where to Look Isn't Obvious; Experience Influences Viewing Strategy

In recent experiments [8, 9], we investigated reading comprehension using various graphical and textual representations of nested conditional structures (Figures 4–7). The experiments included question-answering tasks (corresponding to 'What does this program do?' or 'What made it do that?') and same-different comparisons between programs that allowed us to observe subjects' inspection strategies.

Considerable attention was given to secondary notation in the preparation of the stimuli, both graphical and textual. In all versions, we attempted to maximize structural cues. Layouts were kept as 'clean' as possible, minimizing distractions, and taking account of grouping, alignment, white space, and logical flow. Independent experts were asked to evaluate the layouts, in order to ensure that our interpretation of 'best practice' matched expert usage.

Overall, graphics was significantly slower than text. Moreover, graphics was slower than text in all conditions, and the effect was uniform; the mean time for graphics conditions was greater than the mean time for text for every single subject. The intrinsic difficulty of the graphics mode was the strongest effect observed.

These results were, in general, confirmed by Moher et al. [14] in their replication of these experiments using three forms of Petri net representations. They found no instances in which graphical representations out-performed their textual counterparts. It is interesting to note that Moher and colleagues devised Petri net variants for the purposes of their experiment—in other words, the differences in the graphical representations were purely a matter of secondary notation. Moher et al. observed that "...performance was strongly dependent to the layout of the Petri nets. In general, the results indicate that the efficiency of a graphical program representation is not only task-specific, but also highly sensitive to seemingly ancillary issues such as layout and the degree of factoring"—i.e., secondary notation.



But the most interesting results from our recent experiments lay in the differences in strategies employed by subjects (reported fully in [19]). Strategy differences were strongly related to prior experience; broader experience led to more flexible and more appropriate performance: more effective navigation of structures, match of inspection strategy to the notation and question, and attention to secondary notation.

Novices vs. Experts: Observed Strategy Differences

Novice strategies were more rigid on one hand, and more chaotic on the other. Strategies varied widely among novice subjects. Some novices adhered to much the same strategy regardless of the task. Other novices dithered, chopping and changing among strategies in mid-task, with little apparent consolidation.

Novices suffered from mis-readings and confusions. Their strategies took little obvious account of the structure of the task or of the notation. They appeared often to struggle with a mismatch of strategy and notation, so that they became confused during a reading or they mistrusted their responses.

In contrast, the expert users of graphical notations were more consistent as a group, with different subjects choosing similar strategies. They tended to use the practice trials to identify a suitable strategy for each style of question and then adhere to that strategy for similar subsequent trials. The strategy tended to match the question. When comparable graphical and textual representations were presented side-by-side, experienced readers nearly always used the text to guide their reading of the graphics.

Unlike text, which is always amenable to a straight, serial reading, graphics requires the reader to identify an appropriate inspection strategy. There are few cues to navigation in graphical representations, and some of those few rely on secondary notation. Far from guaranteeing clarity and superior performance, graphical representations may be more difficult to access. Novice users of graphics tend to lack reading and search strategies—to lack proficiency in secondary notation—and tend not to match their strategies to the particular nature of the task or the representation.

Determining What is Relevant; Recognizing Secondary Notation

That novices should display more chaotic behavior and lack strategic range is predictable from the existing literature on differences between novices and experts (e.g., [3]). While novices are distracted by syntax or surface features, experts are better attuned to semantic structures and to the ways in which those underlying structures are cued in the surface layout. The overall pattern is that experts are able to handle information at different levels: they are able both to develop overviews or abstract models and to understand the consequences and significance of detail. What is more interesting here is the evidence that novices and experts differ in what they see and in how they value the available cues; experts apparently base their strategies on different information, for example, *“The shape of the diagram gives information about the content of the diagram, at least for the initiated.”* [emphasis added]

For example, novices typically have difficulty in determining what is important or relevant—in contradiction to the common assumption that graphics makes such information obviously accessible. The less experienced subjects were unable to exploit the secondary notation of the graphical representations which would have improved their reading performance.

Apparently, irrelevance is as much a problem as relevance: a visible symbol is interpreted as a relevant symbol. (cf. Anzai’s observation [1] that novices making drawings for mathematics problem-solving record everything, whereas experienced solvers record only what is needed) The novices were often confused by the explicit connections—if a line existed, it must be relevant. They were uneasy about the completeness of their review; they seemed unable to satisfy themselves that they had read the diagrams thoroughly.

Whereas the less experienced readers were apparently unable to recognize the secondary notational cues available in all of the programs, the experts did recognize and take advantage of sub-term groupings, allowing them to reduce their search space and focus on the portion of the structure containing the relevant information. The secondary notation cues used

to emphasize these groupings allowed experienced readers to see how the variables divided the decision space, so that they could limit their search to the relevant portion of the structure.

While the experts made use of layout clues, they did not confuse layout with logic. Good use of secondary notation was helpful, but changes to layout—

Figure 2. The circles on this ‘novice’ diagram enclose some significant mis-cues:

- *Circle 1:* Although these two components are of the same type, one has been reflected (its inputs and outputs rearranged), giving a misleading impression of symmetry. Their visual symmetry and proximity do not reflect logical relatedness; on the contrary, these components are unrelated in their use.
- *Circle 2:* The extra bends in this wire are probably an editing fossil. Notice also the convoluted wiring at the left of the diagram.
- *Circle 3:* The wires to this gate have two peculiarities: the wire enters Input 1 at an awkward angle, and there are two routes (one redundant) into Input 1.
- *Circle 4:* These two gates have erroneously been made to look like a bi-stable flip-flop. This is a striking misuse of a strong perceptual cue.

```

3a
while ((used!=1) || (a[0] !=1))
{ if (a[0] & 0x1) { k=1; for (c = 0; c <= used; c++)
{ a[c] = 3 * a[c] + k; k = a[c] / 10; a[c] = a[c] % 10;}
if (a[used])
{ used++; if (used >= 72)
{ printf ("Run out of space\n"); exit(1);}}
} else {k = 0;
for (c = used - 1; c >= 0; c--)
{ a[c] = a[c] + 10*k; k = a[c] & 0x1; a[c] = a[c] >>1;}
if (a[used - 1] == 0) used--; }count++;
}

```

```

3b
while ((used!=1) || (a[0] !=1))
{ if(a[0]&0x1)
{ k=1;
for (c=0; c<=used; c++)
{ a[c]=3*a[c]+k;
k=a[c]/10;
a[c]=a[c]%10;
}
if (a[used])
{ used++;
if (used>=72)
{ printf ("Run out of space\n");
exit(1);
}
}
}
else
{ k=0;
for(c=used-1; c>=0; c--)
{ a[c]=a[c]+10*k;
k=a[c]&0x1;
a[c]=a[c]>>1;
}
if (a[used-1]==0) used--;
}
count++;
}

```

Figures 3a and 3b: These fragments of C code (borrowed from an existing commercial application) compile to the same object code. But the introduction of basic secondary notational cues (spacing, indentation, and line breaks) in Figure 3b provides clues for the reader to a structure that may be obscured in Figure 3a.

although possibly inconvenient—were usually recognized as superficial.

It appears that the expert's categorization skills and ability to organize information on the basis of underlying abstractions are reflected in the expert's ability to interpret surface features as clues to an underlying structure. The expert's assimilation of information is not limited by the boundaries of the formalism. Having learned to exploit cues outside what is formally defined, the expert has access to information 'invisible' to non-experts. Hence, experts are distinguished by their acquired abilities to 'see': to perceive as salient the information relevant

to a solution and a task, to choose (or change) their perspective on it, and to choose what not to see—to ignore the inessentials, whether in the search space, in the juggling of constraints, or in a representation.

It is worth noting that expertise cannot compensate for everything. Graphics was uniformly slower than text; it is apparent that even the expert reader of graphical notations is doing a hard job. (Said one expert: *This is hard work... There's no easy way. It's going to be very difficult to explore all this maze in finite time.*) The sight of subjects crawling over the screen with mouse or fingers, talking aloud to keep their working memory updated, was remarkable. One of the distinctions between expert and novice behavior was that the experts made better use of their fingers, occasionally using all ten to mark points along a circuit.

What Makes Experts Expert: Choosing How to Use Secondary Notation

What distinguishes expert designers is not just domain knowledge, but choosing how and when to use that knowledge. The difficulty in characterizing their expertise lies in that ability to choose and the elusiveness of the heuristics used for selection.

Merely capturing an expert's declarative knowledge (however long and detailed a set of declarations is achieved) does not wholly capture his or her expertise. Designing layouts, like designing systems, is an exercise in constraint management, and the heuristics for managing and balancing constraints within a given context are likely to be complex and subtly context-sensitive. Exploiting secondary notation involves deciding what information to emphasize—and at the expense of what other information.

That decision must anticipate the uses to which the representation will be subjected, and it must take into account the competing demands of different uses. Resolving a conflict among constraints may mean choosing which rules to apply among conflicting rules, and which rules to ignore.

Design conventions attempt to reinforce meaningful use of secondary notation, but such conventions are rarely formalized or codified—probably because, although it is possible to enumerate individual 'rules of thumb' (e.g., that functionally-related elements should be clustered together),

these will often conflict in practice, and the heuristics that resolve the conflicts within a particular context are buried deep in experience and expertise and are not easily externalized. Thus, secondary notation may well account for the difficulty in automatic generation of diagrams from code.

Figure 4. An example of Nest-INE, a sequential notation using nested conditionals with extra cues, which was shown to assist both novices and professionals [23].

Even in electronics, a relatively mature discipline compared to graphical programming, the rules of layout are not well codified. For example, Horowitz and Hill, [10], a well-respected text in the field, gives only 6 rules and 14 hints on “How to Draw Schematic Diagrams” (pp. 1056–1057) and even those leave a great deal to the reader, for example, “In general, signals go from left to right; don’t be dogmatic about this, though, if clarity is sacrificed.” The rules are mostly designed to avoid the drawing of hard-to-read figures and misleading figures which look like something else (i.e., the sorts of miscues illustrated in Figure 2). In drawing the stimuli for our reading comprehension experiments, we too attempted to enumerate some rules of practice gleaned from observation of experts. However, we found that the experts who evaluated the drawings, while recognizing the principles we employed, rearranged their priority order and adjusted the drawings accordingly.

Experts ‘in the wild’ use secondary notation in many different ways, and one expert will re-draw another’s diagram in order to make it more meaningful. Yet experts reliably distinguish between novice and expert diagrams, even if the other experts’ conventions differ from their own. Experts apply—and break—the rules in a systematic way attuned to the underlying structure being represented. Within a given representation, they maintain a consistency of application that gives cues to structure and is detectable by other experts.

In explaining their judgments of drawings, experts talk about recognizing such an underlying system of usage—whether or not it is the usage they prefer.

```

if high :
  if wide :
    if deep : weep
    not deep :
      if tall : weep
      not tall : cluck
      end tall
    end deep
  not wide :
    if long :
      if thick : gasp
      not thick : roar
      end thick
    not long :
      if thick : sigh
      not thick : gasp
      end thick
    end long
  end wide
not high :
  if tall : burp
  not tall : hiccup
  end tall
end high
    
```

So Why are Graphical Representations so Appealing?

Given the growing evidence that graphical representations can be harder work and produce poorer performance than textual ones, why are they so appealing?

It may be simply that graphical representations provide an alternative to text. Myers [15] offers a typical description of the attraction: “...graphics tends to be a higher-level description of the desired actions (often de-emphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier even for professional programmers.” (p. 100)

In a recent survey, programmers of all levels were interviewed about their preferences among graphical and textual representations. Graphical representations were described as:

- richer; providing more information with less clutter and in less space
- providing the ‘gestalt’ effect: providing an overview; making structure more visible; clearer
- having a higher level of abstraction, a closer mapping to the problem domain
- more accessible; easier to understand; faster to grasp
- more comprehensible
- more memorable
- more fun
- ‘non-symbolic’; less formal

Richness. Graphical representations appear potentially richer than textual ones. The experts we observed believed that the graphical properties and the secondary notation made electronics schematics and comparable programming notations richer than any textual equivalent and argued that our experimen-

Figure 5. An example of And/Or, a circumstantial notation based on a production system model incorporating Boolean expressions. The conditional structure shown is logically equivalent to the structure shown in Figure 4, with suitable change of labels (e.g., Bellow = Roar).

```

howl:      if honest & tidy & (lazy | sluggish)
laugh:    if honest & tidy & ¬ lazy & ¬ sluggish
whisper:  if honest & ¬ tidy & (nasty & greedy | ¬ nasty & ¬ greedy)
bellow:   if honest & ¬ tidy & nasty & ¬ greedy
groan:    if honest & ¬ tidy & ¬ nasty & greedy
mutter:   if ¬ honest & sluggish
shout:    if ¬ honest & ¬ sluggish
    
```

tal tasks did not exploit this richness. Similarly, it was charged in [14] that "...comparisons of textual and graphical programming is a tricky business, indeed, since it is difficult to match up the contenders for a fair fight. The Gates vs. Do-If comparison employed in [9] appears in retrospect to have pitted a heavyweight against a lightweight." (Compare Figures 7 and 5. 'Do-If' is called 'And/Or' in this article.)

Nevertheless, none of the subjects and none of the experts we have consulted has been able to identify a task that would favor this perceived richness.

'Gestalt' Overview. Graphics may benefit from a 'gestalt' response, an informative impression of the whole that provides insight into the structure.

Because people need help in grasping complex

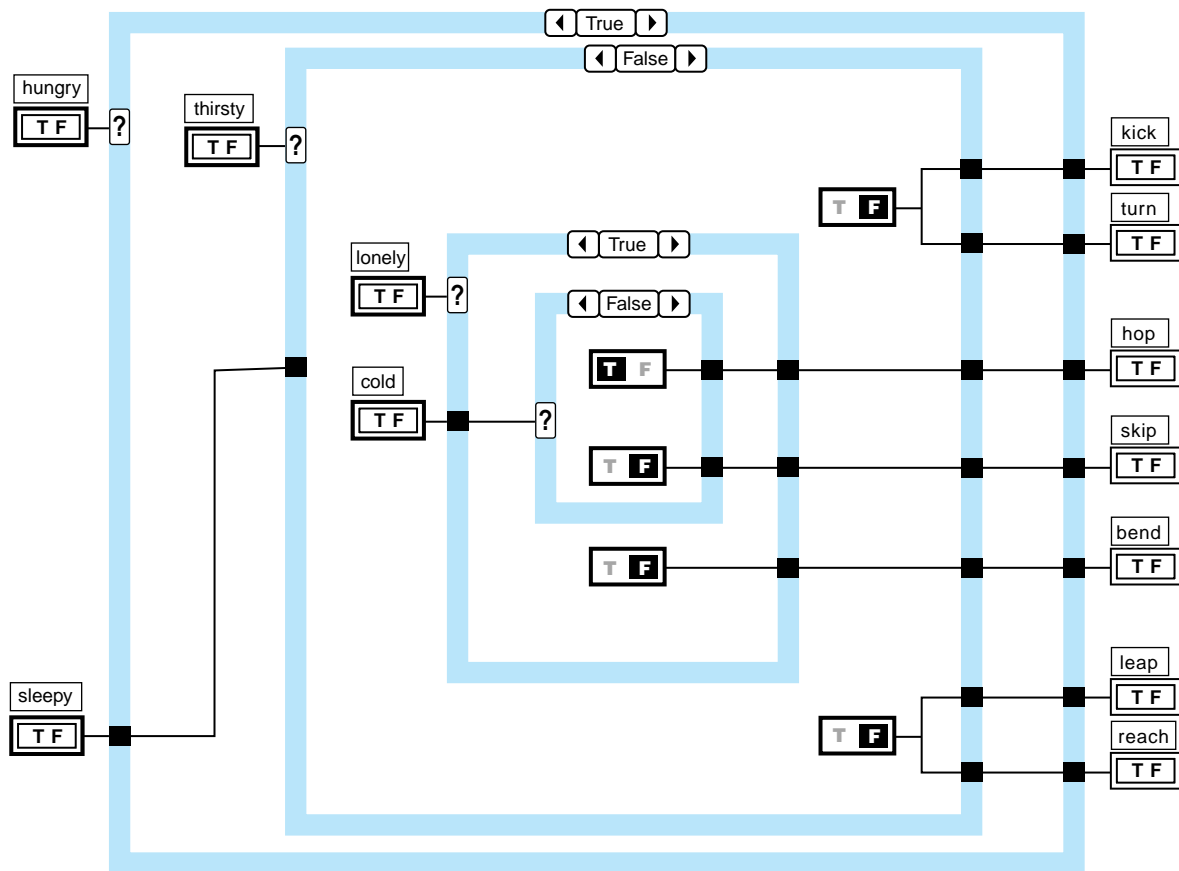


Figure 6. An example of Blocks, a graphical sequential notation. This notation is interactive; mouse clicks on the true/false buttons toggle between the true and false arms of the conditionals. The figure shows the state of the display for Hungry = true, Thirsty = false, Lonely = true, Cold = false. The value of Sleepy is irrelevant in these conditions, although it is relevant in other circumstances (cf. Figure 4, where if High is false, the values of Wide, Deep, etc. are irrelevant). The output is shown by sending a Boolean value to each possible output. In this figure, the output is Hop. The conditional structure shown is both logically and structurally equivalent to the structure shown in Figure 4, with suitable change of labels (e.g., Hop = Roar).

structures quickly, this purported 'gestalt' attribute makes graphics appealing. Yet in the reading comprehension experiments, programmers did not recognize structural similarities among the graphical representations and often found it difficult to compare two graphical programs. In contrast, those programmers who did notice structural similarities among the programs presented were all looking at the *textual* representations.

Mapping to the Domain. Graphical representations appear to offer potential for 'externalizing the objects of thought'—for providing a more direct mapping between internal and external representations by providing representations close to the domain level that make structures and relationships accessible. It was suggested in [12] and [21] that if relations among objects are visually or spatially grasped, it is easier to derive a mental model of a system structure from a

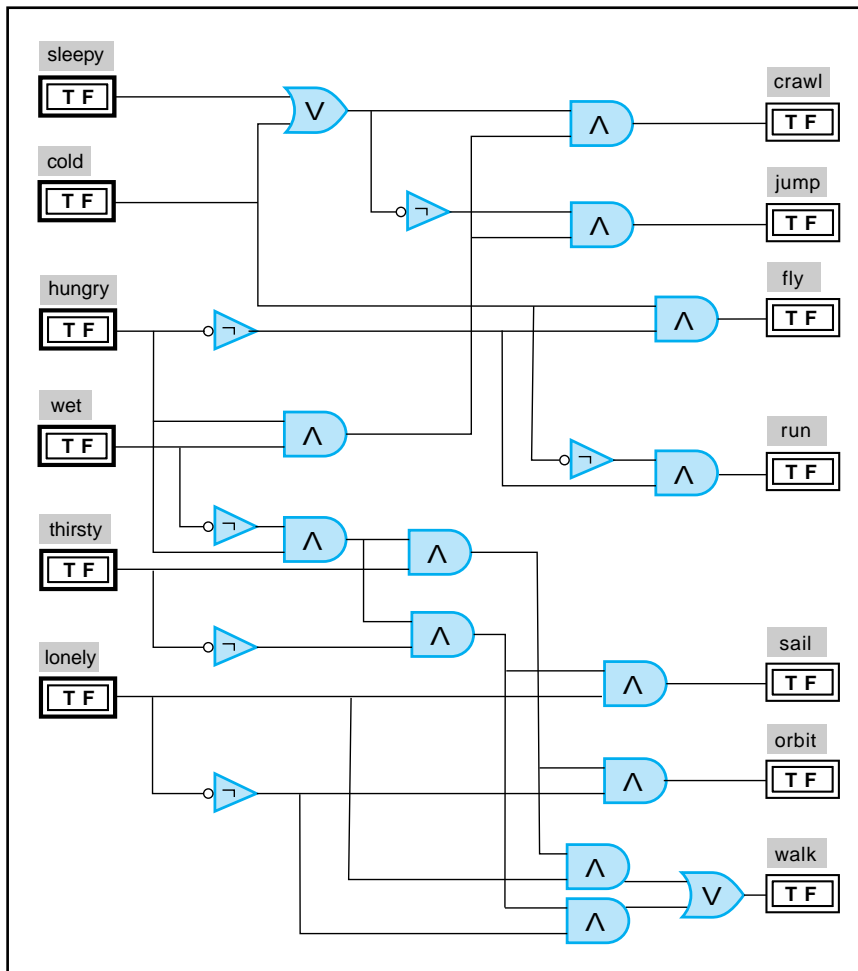


Figure 7. An example of Gates, a graphical circumstantial notation. As in Figure 6, input comes from the left (Sleepy, Cold, etc.) and proceeds to the right. The gates perform the operations of AND (shown as \wedge), OR (shown as \vee), and NOT (shown as \neg). The conditional structure shown is both logically and structurally equivalent to the structure shown in Figure 5, with suitable change of labels (e.g., Orbit = Bellow) and is logically equivalent to the structure in Figure 4, again with suitable change of labels (e.g., Orbit = Roar).

Fun. Graphical representations may just seem more fun or gratifying; the fact that secondary notation is ‘outside the rules’ allows the programmer more freedom to ‘play around’ with layout. The overheads of planning layouts may not matter if there is satisfaction simply in the tinkering required.

graphical representation than from a textual one. But meeting that potential is a challenge.

Experience in digital electronics warns that a typical novice error is too ‘literal’ a transcription of the domain, a failure to abstract; novices often reflect the eventual physical layout rather than the logical layout. In effect, they draw a picture of the artifact, rather than depict the structure of the solution.

Accessibility and Comprehensibility. It may be that an analog representation appears more accessible than a descriptive one—a notation incorporating pictures may seem less daunting than one comprising abstract symbols. A representation that exploits perceptual cueing takes advantage of the abilities of the human visual system. The analog quality appeals to mundane experience, making the notation appear less esoteric.

Cynically, this is a variation of the ‘Cobol effect’ familiar in the history of programming languages: because the vocabulary (in this case the component shapes) looks familiar and ordinary, novices believe they can understand the program. Yet graphical representations can take longer to read and understand, and they are often misunderstood by novices, who cannot ‘see’ the available cues.

The importance of sheer likeability should not be underestimated; it can be a compelling motivator. In general, affect may be as important as effectiveness. The *illusion* of accessibility may be more important than the reality.

Examining the Role of Graphics in Programming Notation

The important difference between ‘textual’ and ‘purely graphical’ seems to be the trade-off between ‘descriptive’ and ‘analog’ representation.

Text, a descriptive representation, derives precision of expression from a small, fixed vocabulary, and it achieves range of expression by regular combination of vocabulary elements. Similarly, readers learn rules of interpretation, so that plain text is read serially (although it may be accessed at random), and text is easily ordered and searched. Plain text doesn’t rely on perceptual responses particular to a sensory mode; text is easily translated from the visual to other modes, as by reading aloud.

Pure graphics, an analog representation, gains from the mapping of perceptual cues to information (e.g., the association of color with type, or of size with number). It may draw on an unlimited vocabulary.

Graphics may benefit from a ‘gestalt’ response, an informative impression of the whole that provides insight into the structure, but it lacks the precision of text, because much information is intrinsic in the analog mappings. The rules of interpretation are not as clearly defined as for text, and so graphics may suffer from ambiguity of interpretation. Offering few cues to navigation, graphics requires the reader to identify some appropriate inspection strategy.

Text is essentially graphics with a very limited vocabulary. Each character is a pure graphic, with perceptu-

ally-learned associations between objects and functions, holds up when the icon represents a concrete object, but Rohr [21] showed that this breaks down when icons represent abstractions. And the advantages of analog properties may diminish with size and complexity. There is a swamping effect: when many icons are used, discriminability decreases [9, 24].

Similarly, secondary notation is often poorly exploited by graphical programming systems. Some (e.g., Prograph) encourage encapsulation into many small units, so that the layout is too fragmented to benefit much from structure cues. For others (e.g., LabView), even for small programs, the layout often becomes so well-filled that little can be done to re-arrange it to convey additional information. The possibility of using layout in a controlled and useful way is dominated by the need to keep the visual picture reasonably clean; for example, placing components to minimize crossings of connection lines takes priority over placing them adjacent to functionally-related components.

Furthermore, freedom to make use of the secondary notation potentially available may carry high overheads, requiring users to plan the layout of the programs as well as the algorithms, for example, *I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it.*

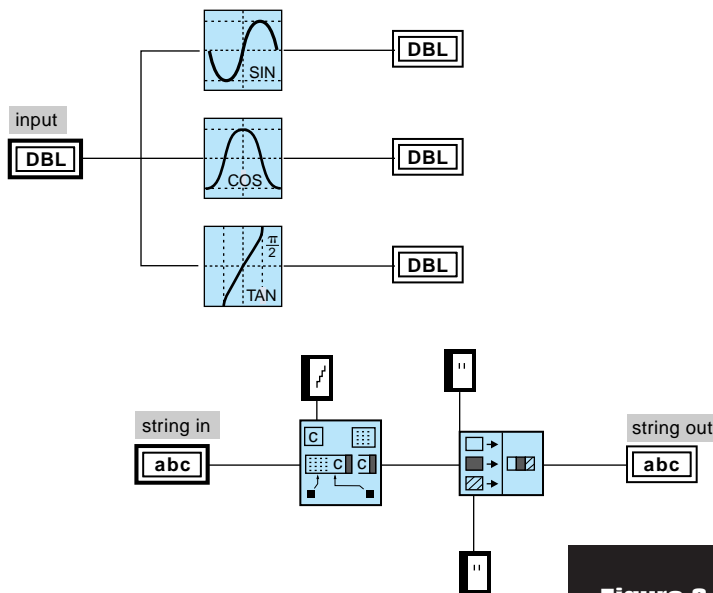


Figure 8. These tiny fragments of LabView code show the use of iconic notational elements that give pictorial clues to their functions.

al qualities, but readers have learned to see characters quite differently, as ‘non-graphical’ text symbols. The plain text system has, through a process of abstraction and evolution, suppressed the perceptual qualities of the individual graphics that comprise it. Further, the learned associations with text disrupt automatic perceptual processing.

Instead, text—conventional, natural-language, printed text—has had several centuries in which to evolve into a well-tuned medium for the conveyance of technical information.

In some ways, many of the current ‘graphical’ programming languages—many of them little more than box-and-line systems—are more ‘textual’ than ‘purely graphical’; they have replaced the familiar ASCII character set with an alternative fixed vocabulary of symbols (often a perceptually-limited repertoire of rectangle variants) and have not taken advantage of the analog mapping.

Some graphical programming systems (e.g., LabView, see Figure 8) provide a more analog, iconic vocabulary, so that a component’s appearance gives clues to its meaning. The notion that icons have the advantage of built-in mnemonics, making use of pre-

Conclusions Graphical Readership is an Acquired Skill

Novice and expert readers appear to notice and concentrate on different graphical details—to ‘see’ differently. Whereas novices tend to confuse visibility with relevance, experts take advantage of secondary notation cues to enable them to recognize sub-term groupings, to match patterns (which novices do not recognize) and to dis-

regard irrelevant information. The programmer learns to take advantage of available perceptual cues and to apply acquired rules of interpretation.

The correlation between experience and reading strategy is not exclusive to graphics. In a previous experiment on the reading of programs, the effects for Pascal programmers did not apply to Basic programmers, interpreting the difference as a difference between notations [7]. However, it was shown in [5] that the results were caused by differential training backgrounds: when Basic programmers were taught the precepts of structured programming, the differences disappeared. What a reader sees is largely a matter of what he or she has learned to look for.

The skills of graphical readership, both perceptual and interpretive, are learned skills. This accords with



research on perception (e.g., [6, 25]), which emphasizes the role of relevance in visual processing: perceptual skill means learning to discriminate relevant features and to synthesize them into a meaningful whole. Studies like Koga and Groner's study of non-Japanese subjects learning Kanji characters [11] show that eye movements and scanning behavior change as cognitive skills are acquired.

Knowing what to expect, where to look, and what to look for—the cognitive components of an inspection—affects the strategies the individual employs in reading an information structure, and increasing expertise is evidently reflected in changes of perceptual strategy. Training and experience play a significant role in determining what is salient.

Experts and Novices have Different Notational Needs

For programming, one might conjecture that we should give experts languages that have good scope for secondary notation, because they are likely to be creating complex programs that will need to be understood by others (or by themselves at a later date) and so would benefit from enriched cueing.

Novices, on the other hand, might benefit from a more constrained system in which secondary notation is minimized, in order to reduce the richness and the potential for mis-cueing and misunderstanding.

This conjecture is apparently at odds with more familiar arguments, in which graphical programming is usually advocated for less skilled groups, for novices learning to program or for 'end-user' or 'casual' programmers. This article does challenge the advocacy—on the grounds of accessibility—of graphical representations for novices, because the available evidence suggests that less skilled groups are precisely those less likely to benefit from the secondary notation that enhances access, because novices are unlikely to have the necessary readership skills.

Yet this article also recognizes the appeal of graphical representations, especially for those who have not developed the readership skills for formal representations. The conflict is not unresolvable: the answer still lies in distinguishing between expert and novice skills, and hence in designing systems for novices that embody a disciplined and insightful use of secondary notation in a way that provides appropriate constraints for the novices who will use it. Using graphics to pro-

vide fun doesn't necessitate a graphical free-for-all.

It is essential to move away from the superlativist claims in the literature and toward a recognition that graphics requires readership—and production—skills in the same way that text does, and to provide support in the environment and in the culture for the acquisition and exercise of those skills. Graphical representations, especially those used in notation, far from being intuitively obvious, may require more training from both the originator and the reader to achieve the most effective communication—but may prove richer for expert users. The challenge for the designer is to make good use of secondary notation, 'good' in supporting access to relevant information, 'good' in avoiding mis-cueing, 'good' in matching the secondary notational conventions used by experts in the domain under study, and 'good' in nurturing effective graphical readership.

Accept the 'Bad' with the 'Good'

The hope is that 'good' use of secondary notation can direct attention to relevant information; but 'goodness' relies on individual skill and insight, and readers must learn to recognize the cues. Surprisingly few proponents of graphical superiority have identified what kind of graphics are suitable for particular tasks, even though there are some empirically validated metrics in the literature. The range of accuracy in human perception when using seven different pure graphical techniques for displaying quantitative information was shown in [4]. This was extended to compare thirteen different techniques in perceptual accuracy for quantitative, ordinal, and nominal data in [13]. Such analyses must be extended to describe graphical notations.

'Good' graphics usually means linking perceptual cues to important information, which means both identifying and capturing what is important, and guiding the reader with appropriate cues. It is time to recognize the impact of 'bad' graphics—of haphazard use of perceptual cues and secondary notation—mis-cueing, misleading, misreading, and misunderstanding.

It appears that graphical notations can have a greater capacity to 'go wrong' than textual notations—a graphical representation with distorted secondary notation can cause more confusion than a textual notation which misuses secondary notation. The giddy intertwining connections of a too-complex or poorly designed boxes-and-lines-style representation makes vivid the notion of 'spaghetti code'. Text devoid of secondary notational cues is still comprehensible with an amount of work—moreover, textual programs can be formatted or reformatted by algo-

'Good' graphics usually means linking perceptual cues to important information,
*which means both identifying and capturing what is important,
and guiding the reader with appropriate cues.*


rithm, either by rote or by machine, into something comprehensible. ‘Pretty printers’ of structured languages are familiar tools. But formatting a graphical representation into something comprehensible is not currently reliably possible by algorithm. Even the best examples of ‘graphical pretty printers’ are satisfactory only in highly constrained domains.

In constraint lies the key: ‘good’ secondary notation involves disciplined and appropriate application of constraints to the available freedoms of presentation. Text is by nature more constrained; its linear character provides clues for the reader even when the secondary notation is dysfunctional. But the ‘flip side’ of the greater freedom afforded by graphical representations is the greater potential for mis-cueing and confusion.

Even a well-evolved notation is vulnerable to weaknesses in individual expressive skill. Graphical representations share this problem with textual ones: quality is not guaranteed. There are bad diagrams as well as good ones, just as there are both bad and good textual representations, and there are more or less gifted practitioners as well as more or less experienced ones. The power of secondary notation can be expressive—or expensive.

There is no single panacea. But if we can identify the particular strengths of different sorts of representation, we will be able to design appropriate solutions with a full repertoire of notational options.

Acknowledgments.

Thomas Green was a partner on most of this research and a partner in the genesis of the ideas presented here. Blaine Price and Peter Eastty helped with provocative discussions. The anonymous reviewers made helpful comments. None of these observations would have been possible without the generous participation of the expert programmers and designers. 

References

1. Anzai, Y. Learning diagram-drawing and graphical reading skills: analysis and design. Presented to: *Workshop on Graphical Representations, Reasoning, and Communication, AFED '93* (Edinburgh, August, 1993).
2. Bouwhuis, D. G. Reading as goal-driven behaviour. In B. A. G. Elsendoorn and H. Bouma, Eds., *Working Models of Human Perception*. Academic Press, 1988, pp. 341–362.
3. Chi, M.T.H., Glaser, R., and Farr, M.J., Eds. *The Nature of Expertise*. Erlbaum, Hillsdale, NJ, 1988.
4. Cleveland, W.S., and McGill, R. Graphical perception: theory, experimentation and application to the development of graphical methods. *J. Amer. Statistical Assoc.* 79, (1984), 531–554.
5. Davies, S. P. Skill levels and strategic differences in plan comprehension and implementation in programming. In A. Sutcliffe and L. Macaulay, Eds., *People and Computers V*. Cambridge University Press, Cambridge, UK, 1989.
6. Gibson, E.J., and Levin, H. *The Psychology of Reading*. MIT Press, Cambridge, Mass., 1975.
7. Gilmore, D. J., and Green, T. R. G. Programming plans and programming expertise. *Quarterly J. of Experimental Psych.* 40A (1988), 423–442.
8. Green, T.R.G., and Petre, M. When visual programs are harder to read than textual programs. In *Proceedings of the Sixth European Conference on Cognitive Ergonomics (ECCE-6)*, Budapest, Hungary, September 1992.

9. Green, T. R. G., Petre, M., and Bellamy, R. K. E. Comprehensibility of visual and textual programs: A test of Superlativism against the “match-mismatch” conjecture. In *Empirical Studies of Programmers Fourth Workshop*. Ablex, 1991.
10. Horowitz, P., and Hill, W. *The Art of Electronics*. 2d ed. Cambridge University Press, Cambridge, UK, 1989.
11. Koga, K., and Groner, R. Intercultural experiments as a research tool in the study of cognitive skill acquisition: Japanese character recognition and eye movements in non-Japanese subjects. In H. Mandl and J.R. Levin Eds., *Knowledge Acquisition from Text and Pictures*. North-Holland, 1989, 279–291.
12. Kosslyn, S.M. Imagery and internal representation. In E. Rosch and B.B. Lloyd, Eds., *Cognition and Categorization*. Erlbaum, Hillsdale, NJ, 1978, 227–286.
13. Mackinlay, J. Automating the design of graphical presentations of relational information. *ACM TOGS* 5, 2 (1986), 110–141.
14. Moher, T.G., Mak, D.C., Blumenthal, B., and Leventhal, L.M. Comparing the comprehensibility of textual and graphical programs: The case of Petri nets. In C.R. Cook, J.C. Scholtz, and J.C. Spohrer, Eds., *Empirical Studies of Programmers: Fifth Workshop*. Ablex, 1993, 137–161.
15. Myers, B.A. Taxonomies of visual programming and program visualization. *JVLC* 1, 1 (1990), 97123.
16. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, (1987), 295–341.
17. Petre, M., and Green, T. R. G. Where to draw the line with text: Some claims by logic designers about graphics in notation. In *INTERACT'90 Conference on Computer-Human Interaction*, 1990.
18. Petre, M., and Green, T. R. G. Requirements of graphical notations for professional users: Electronics CAD systems as a case study. *Le Travail Humain* 55, 1 (1992), 47–70.
19. Petre, M., and Green, T.R.G. Learning to read graphics: some evidence that ‘seeing’ an information display is an acquired skill. *Journal of Visual Languages and Computing* 4 (1993), 55–70.
20. Raymond, D. Characterizing visual languages. In *The 1991 IEEE Workshop on Visual Languages*, (1991) IEEE Computer Society Press.
21. Rohr, G. Using visual concepts. In S-K. Chang, T. Ichikawa and P.A. Ligomenides, Eds., *Visual Languages*. Plenum Press, 1986.
22. Shu, N.C. *Visual Programming*. Van Nostrand Reinhold, 1988.
23. Sime, M.E., Green, T.R.G., and Guest, D.J. Scope marking in computer conditionals—A psychological evaluation. *IJMMS* 9, (1977) 107–118.
24. Swigger, K.M., and Brazile, R.P. An empirical study of the effects of design/documentation formats on expert system modifiability. In J. Joenemann-Belliveau, T. Moher, and S. Robertson, Eds., *Empirical Studies of Programmers: Fourth Workshop*, Ablex, 1991.
25. Treisman, A. Features and objects in visual processing. *Scientific American* (1986), 106–115.

About the Author:

MARIAN PETRE is Senior Research Fellow and Director of the Centre for Informatics Education Research in the Faculty of Mathematics and Computing at the Open University (formerly Research Fellow in the Open University’s Institute of Educational Technology). **Author’s Present Address:** The Open University, Walton Hall, Milton Keynes, MK7 6AA, U.K. email: m.petre@open.ac.uk

LabView is a trademark of National Instruments, Inc. Prograph is a trademark of TGS Systems Ltd

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.