Paper accepted for publication in ***Artificial Intelligence Review***, special issue on *Thinking with Diagrams*.

Cognitive Factors in Programming with Diagrams

Alan F. Blackwell
Computer Laboratory
University of Cambridge
Cambridge, U.K.
alan.blackwell@cl.cam.ac.uk

Kirsten N. Whitley
Department of Computer Science
Vanderbilt University
Nashville, Tennessee, U.S.A.
whitley@computer.org

Judith Good
Human Communication Research Centre
University of Edinburgh
judithg@cogsci.ed.ac.uk

Marian Petre
Centre for Informatics Education Research
Faculty of Mathematics and Computing
Open University
Milton Keynes, U.K.
m.petre@open.ac.uk

Abstract

Visual programming languages aim to broaden the use of diagrams within the software industry, to the extent that they are integrated into the programming language itself. As a result, they provide an ideal opportunity to study the benefits of diagrams as an external representation during problem solving: not only is programming a challenging problem-solving activity, but the effect of diagram usage can be directly assessed by comparing performance while using a visual programming language to performance with a standard textual language. There have been several misconceptions amongst visual language researchers regarding the role of diagrams in software design, but these are being addressed by empirical studies and by new theories of notation design derived from studies of visual programming. Based on this research, the authors are able to recommend several new directions for research into thinking with diagrams.

Keywords: diagrams, diagrammatic reasoning, visual programming, psychology of programming

1. Introduction

This paper investigates a specific class of diagrammatic notations – Visual Programming Languages (VPLs) – considering them from the perspective of research into psychology of programming. This field has developed a set of research methods specifically aimed at measuring human performance in solving problems with various external notations. As a result, the psychology of programming work dealing with visual programming should offer important insights into the application of diagrams in problem solving contexts.

This paper starts with an overview of VPLs – the reasons for their introduction, and the role that they play within software development. It then gives a brief introduction to research methods that are used in psychology of programming. We review the empirical evidence for benefits resulting from diagram use in programming, then describe new insights into the general properties of notations that have been derived from this empirical work. Finally we propose some of the most pressing topics for ongoing research into diagrammatic properties of VPLs.

2. The Nature of Visual Programming

Computer programming is a challenging intellectual task, involving complexity equivalent to other design and engineering activities. In the term introduced by Reitman (1964) it belongs to the class of "ill-structured" problems that cannot be solved by a strictly defined procedure.

At the same time, programming has two characteristics that make it apparently very appropriate as a research context for studying diagrammatic reasoning. Firstly, many programming activities take place within a technically constrained environment. This makes it possible to observe, record and measure many (but not all) of the actions taken by a programmer. Secondly, diagrams are often associated with computer programming, whether as an external representation employed as a private aid to thought, or as a communication medium between members of a software project team.

Like other engineering design professionals (see Ferguson 1992 for a review), programmers commonly create informal and ephemeral diagrams, both privately and as "talking sketches". As with other engineering disciplines, many of these informal conventions have been formalized in a way that makes them more persistent and suitable for maintenance as a form of design documentation. In programming, furthermore, those that have been formalized are readily amenable to automated translation and immediate testing.

Unlike most diagrams used in other engineering disciplines, software diagrams are not constrained by the need for graphical correspondence to the physical shape of any designed artifact. This has resulted in relatively greater freedom regarding the form of the diagram elements. Also unlike the representations used in other design disciplines, new software diagrams are continually being developed, so it is possible to observe the effect of successive generations of diagram conventions, or even influence the development of future generations of design notation.

VPLs have evolved from diagrammatic notations such as the relatively familiar flowchart. Many of these notations prescribed the behaviour of a program to the same level of detail that a textual programming language would do. When design diagrams started to be drawn directly on the screen of the computer, they could then be translated automatically into equivalent lines of program text. The diagram then becomes part of the program.

There have been at least a hundred proposals for VPLs. Glinert (1990a, 1990b) has edited a two volume collection of landmark papers in this field, dating from 1975. More recent research developments are reported in the proceedings of the now annual IEEE symposia on visual languages, and in the Journal of Visual Languages and Computing. Taxonomies of VPLs (as well as of visualization systems) have been proposed by Myers (1990) and by Price, Baecker & Small (1993).

The boundaries dividing VPLs from other classes of diagram are not always clear. We take the view that a programming language must be executable - it must have precise semantics defined in terms of expected changes in state of a digital computer. In this paper we do not discuss products such as Microsoft Visual Basic and Visual C++, where the logical behaviour of the program is fully expressed in the form of a textual programming language rather than by diagrams.

3. Psychology of Programming

Research into the cognitive processes involved in computer programming is derived from general cognitive theories of problem solving after Newell and Simon (1972). Some notable early cognitive theories of programming include those of Weinberg (1971) and Shneiderman (1980). Many of the aspects of these models will be familiar to cognitive scientists. They include separate processing of syntactic and semantic information, the collection of expert knowledge into chunks, the structuring of regularly-used information into schemas, and the solution of design problems in terms of previously acquired and frequently modified plans.

An overview of the range of research methods currently used in the psychology of programming can be found in the collection edited by Hoc, Green, Samurçay and Gilmore (1990). This volume also includes a chapter by Gilmore specifically reviewing and advising on empirical research methods that are suited to investigations of programming.

The predominant research technique in psychology of programming adopts the hypothesis testing methods of experimental psychology. Studies generally involve observation of small groups of subjects performing the same constrained, well-defined tasks. Performance might be measured in terms of the time required to complete a simple task, accuracy of response, or classification of statements found in a verbal transcript from the experimental subject. Individual differences between the subjects are not analysed in detail; instead the experimenter compares the differences between groups of subjects, looking for statistically significant variations in performance that was expected to result from the treatment assigned to each group.

Partly as a result of this, psychology of programming, like experimental psychology, has been accused of not addressing the concerns of "real" programmers who deal with messy, ill-defined problems where the main challenges are those of collaboration with customers and colleagues over long periods of time. The focus on statistical analysis also obscures individual differences in strategies used for specific tasks. This is of particular concern where those strategic differences may have far more effect on performance than the experimental hypotheses that are being tested.

There are alternatives to the pure hypothesis-testing approach, including empirical studies that aim to gather qualitative data about the processes involved in programming, as well as longitudinal field studies that chart the course of learning a programming language or observations of larger-scale software development projects over time.

4. Empirical Studies of Diagrammatic Notations Used in Programming

Relatively few empirical studies have focused directly on the use of diagrammatic notations in programming, although recent activity within the visual programming community may reflect growing interest in undertaking more such studies. This section summarizes existing studies, both to show the range of questions investigated and to show common themes in the findings. This summary includes studies on flowcharts, as well as research on VPLs. Interested readers can consult (Whitley 1997a) for a more detailed and extended discussion.

4.1. Flowcharts

More studies have focused on flowcharts than on other visual programming constructs. To date, these suggest that flowcharts can outperform text for certain tasks but not for the entire programming process. Scanlan (1989) investigated the comprehensibility of structured flowcharts and textual “pseudo-code” as representations for conditional logic. He asked programming students to view conditional logic and then answer questions about the states required to trigger given actions. Flowcharts had a significant advantage for the time needed to comprehend an algorithm, but also in other variables such as response accuracy. These effects were observed in simple cases as well as in complex ones. Scanlan’s results suggest that flowcharts can have a beneficial effect for certain tasks; in particular, they illuminate the control-flow of conditional logic. Similar results have occurred in studies of programming students by Vessey and Weber (1986) and by Cunniff and Taylor (1987).

In contrast, no studies have shown flowcharts having a practical advantage over text across the full range of programming activities. Curtis et al. (1989) studied the flowchart in its historical role as a notation supplemental to textual code (i.e., as a form of design and/or program documentation). They identified two dimensions (symbology and spatial arrangement) capable of categorizing a wide range of notations. The symbology dimension measures the succinctness of a notation and includes three possibilities: unconstrained text (natural language), constrained text and ideograms. The spatial arrangement dimension captures the extent to which a notation's layout highlights the execution paths (in the case of flowcharts, the control flow) of a program; this dimension also has three values: sequential, branching and hierarchical. Combining all possibilities yields nine

documentation formats, which were tested in comprehension, coding, debugging and modification tasks. Participants in these experiments were professional Fortran programmers.

Consistent effects of symbology were found in comprehension, coding and debugging. Prose was the most ineffective symbology format for most tasks, whereas constrained language and ideograms were almost equivalent. As for spatial arrangement, only small effects were observed; these occurred in situations in which control flow information was a factor in the task. These results are largely due to the branching arrangement, which improved performance in some tasks where control flow was important. Putting symbology and spatial arrangement together, the constrained/sequential representation typically outperformed the other forms of documentation. In sum, Curtis et al.'s results are consistent with Scanlan's in that the branching arrangement did help in tasks that emphasized control flow; but overall the constrained/sequential representation was equal or better for most tasks.

4.2. Studies of VPLs in current use

There are few empirical studies of VPLs that are in current use. A study by Pandey and Burnett (1993) stands out as the strongest controlled study favorable to VPLs. This tested performance on matrix problems using two textual programming languages (Pascal and a modified form of APL) and a diagrammatic subset of the research VPL Forms/3. 73% of Forms/3 solutions were completely correct compared to 53% of APL solutions and 40% of Pascal solutions. Their results apply to their language and programming environment as a whole; further study would be required to pinpoint the impact of their particular visual representation.

Aside from this study, the majority of the empirical studies have focused on the few commercially-available VPLs, most notably Pictorius Prograph, National Instruments LabVIEW, and Hewlett Packard VEE. Prograph is the only one of these that is promoted as a general purpose programming language – LabVIEW and VEE are frequently described as measurement and control languages, and characterized as accessible to scientists and engineers having limited programming experience. The visual syntax of LabVIEW and VEE expresses data flow in a way that resembles electronic circuit diagrams (this is particularly explicit in LabVIEW).

Baroth and Hartsough (1995) report on their experience using both LabVIEW and VEE. One case study compared two teams developing the same system in parallel. One team used the textual language C, while the other worked in LabVIEW. At the end of the three-month project, the LabVIEW team had made far more progress. From this study and more than 40 other projects, Baroth and Hartsough enthusiastically report performance benefits for both LabVIEW and VEE. They attribute the productivity gains to increased communication between the customer and developer, which arises, they say, from the visual syntax of the VPLs. Their customers were engineers and scientists comfortable with circuit diagrams, so the circuit-like syntax helped them understand the program.

In contrast, the one controlled laboratory study of LabVIEW seemingly contradicts Baroth and Hartsough's speculations about visual syntax. Green, Petre and Bellamy pitted LabVIEW's two forms of visual conditional logic against two textual notations (Green, Petre & Bellamy 1991, Green & Petre 1992). They started from Gilmore and Green's match-mismatch hypothesis, which states that problem-solving performance depends on whether the structure of a problem is matched by the structure of a notation (Green 1977). This was first established using two textual forms of conditional logic: an "if-then-else" notation facilitates answering forward questions (i.e., "which action results for given conditions?") while a "do-if" notation facilitates answering backward questions (i.e., "which conditions must exist to evoke a specified action?"). LabVIEW happens to provide two notations for expressing conditional logic. Green, Petre and Bellamy proposed that these correspond respectively to a forward and a backward form, and therefore compared them to forward and backward questions using two textual notations. The LabVIEW notations did exhibit the expected match-mismatch effect, but the effect size was less than anticipated. Instead response times were twice as long in comprehension questions using the visual notations.

In subsequent articles, Green and Petre (Petre 1995; Petre & Green 1993) argue that "secondary notation" accounts for a large part of the (un)readability of a visual notation, and that the ability to read a visual notation and its associated secondary notation is dependent upon training. Secondary notation – the use of layout and other informal cues to express structure – is defined below along with Green's other "cognitive dimensions of notations". Green and Petre base their argument on observational studies of novice LabVIEW programmers and expert electronics designers at work. As a group, the experts approached the study questions fairly consistently; they tended to use like strategies and to choose strategies based upon the style of the problem. In contrast, LabVIEW programmers were inconsistent in their strategies, even to the point of changing strategies in mid-task. Green and Petre attribute this to relative inexperience. Also, whereas the electronics experts recognized and took advantage of spatial groupings, the LabVIEW programmers seemed unaware of this secondary notation.

The Green and Petre results have been further explored by Moher et al. (1993) in a study that compared text to petri nets. Moher et al. were interested in whether petri nets show promise as the visual basis for a VPL. They used the same experimental design and comprehension questions employed in the Green, Petre and Bellamy study, but different visual representations. They designed three different petri net notations: one corresponded to the if-then-else statement (a forward form), one to LabVIEW's gates notation and one to a textual do-if statement (both backward forms). The Moher et al. study confirmed the match-mismatch hypothesis for textual notations but not for petri nets. Petri nets were faster for backward than for forward questions, but two of them performed worse than their text counterparts, while the third was not significantly different from text. These results concur with Green, Petre and Bellamy that the match-mismatch hypothesis cannot account for all of the differences seen in this experiment.

4.3. Algorithm visualization

A handful of empirical studies have investigated the use of animated pictorial visualization of software algorithms as a teaching tool. Often, such studies fall under the purview of diagrammatic research, for example when the visualization is based on a node and arc graph. Despite popular enthusiasm for the concept of visualization, these studies have found no conclusive evidence to recommend its use. For example, Stasko, Badre and Lewis examined the effects of visualizing a heap algorithm (Stasko, Badre & Lewis 1993). There was no significant difference in comprehension between students who learned the algorithm via a text description and those who also saw a visualization. In view of this failure of empirical validation, Gurka and Citrin (1996) advocate a careful meta-analysis of earlier studies, looking especially for factors that might have produced false negative results.

4.4. Status of empirical studies

The existing studies of diagrammatic notations used for programming tasks have contributed examples in which diagrammatic notations resulted in performance benefits. Several have shown visual notations outperforming text in either time or correctness, sometimes in both. Yet many basic assumptions surrounding these studies have not been investigated:

- Diagrams are likely to be better than text for some problems, worse for others. The differences lie in the costs of locating and indexing information, as well as in differences of cognitive processing of symbolic and spatial information – but these are open questions.
- Experts appear to do things differently from novices, but in many ways that are extremely hard to analyse. There are differences in processes of identification, indexing, selection, abstraction, matching strategy to task, finding ways of making complex problems tractable, and so on.
- Every notation makes some information accessible at the expense of obscuring other information. Hence the match-mismatch hypothesis.
- Despite the fact that differences between subjects are not analyzed in detail, individuals do differ.
- When researchers make assumptions about the relationships between programming languages and reasoning, they often don't hold up to empirical scrutiny of how people really program.

This section has raised more questions than it has answered; we don't wish to disguise the fact that many research questions in psychology of programming are both open and difficult. Nevertheless, we consider that these questions are central to the understanding of diagram usage in visual programming – a field where the properties of diagrams are central.

5. Cognitive Dimensions of Notations

The study of the cognitive factors involved in VPLs can be traced to Fitter and Green (1979). Green (with many collaborators, including several of the present authors) remains a central figure in this field. Despite this long history, empirical studies of programmers have had little effect on the design of new programming languages. They have generally addressed quite detailed aspects of programming style or language features, and provide grounds for a critique of specific language features rather than the broader issues of language format. Little has changed since the complaint made by Shneiderman in 1980 that "Computer scientists … make broad claims for the simplicity, naturalness, or ease-of-use of new computer languages or techniques, but do not take advantage of the opportunity for experimental confirmation" (Shneiderman 1980, p. xiii).

Green (1989, 1991; Green and Petre, 1996) has introduced the “cognitive dimensions of notations” framework as discussion tools – descriptions of the artifact-user relationship – intended to raise the level of discourse. (The following description of cognitive dimensions summarizes a more complete treatment in Green & Petre, 1996).

Cognitive dimensions constitute a small vocabulary of terms describing the cognitively-relevant aspects of structure of an information artifact, and show how they can be traded off against each other. Any cognitive artifact can be described in these terms and, although that description will be at a very high level, it will predict some major aspects of user activity. The framework is task-specific, concentrating on processes and activities rather than the finished product. This broad-brush framework supplements the detailed and highly specific analyses typical of contemporary cognitive models in HCI.

5.1. Partial list of cognitive dimensions

The framework of cognitive dimensions consists of a small number of terms which have been chosen to be easy for non-specialists to comprehend, yet capture a significant amount of the psychology and HCI of programming. The so-called ‘dimensions’ are meant to be coherent with each other, like physical dimensions. A partial list of dimensions follows, with thumb-nail descriptions:

Abstraction gradient: An abstraction is a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure. What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

Closeness of mapping: What ‘programming games’ need to be learned? Programming requires a mapping between a problem world and a program world. The closer the programming world is to the problem world, the easier the problem-solving ought to be.

Consistency: When some of the language structure has been learned, how much of the rest can be inferred successfully?

Diffuseness: How many symbols or graphic entities are required to express a meaning? Some notations use a lot of symbols or a lot of space to achieve the results that other notations achieve more compactly.

Error-proneness: Does the design of the notation induce ‘careless mistakes’? Does it make them hard to find once they have occurred?

Hard mental operations: Are there places where the user needs to resort to fingers or pencil annotation to keep track of what's happening?

Hidden dependencies: 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. Is every dependency overtly indicated in both directions?

Premature commitment: Do programmers have to make decisions before they have the information they need?

Progressive evaluation: Can a partially-complete program be executed to obtain feedback on ‘How am I doing?’ The ability to evaluate their own problem-solving progress is essential for novices and desirable even for experts.

Role-expressiveness: Can the reader see how each component of a program relates to the whole? Role-expressiveness is enhanced by meaningful identifiers, by well-structured modularity, and by the presence of `beacons' that signify certain code structures.

Secondary notation: Can programmers use layout, choice of naming conventions, grouping of related statements, colour, and other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?

Viscosity: How much effort is required to perform a single change? One standard example of viscosity is having to make a global change by hand because the environment contains no global update tools.

Visibility: Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will?

5.2. Trade-offs among dimensions:

The purpose of the cognitive dimensions framework is to lay out the cognitivist's view of the design space in a coherent manner, and where possible to display some of the cognitive consequences of making a particular

bundle of design choices. From the point of view of the designer, there are important trade-off relationships between the cognitive dimensions. Changing the structure of a notation to reduce viscosity, for example, is likely to affect other dimensions (perhaps by introducing hidden dependencies or increasing the abstraction gradient). Far more analysis of trade-off relationships needs to be done. What is important is to bear in mind that because these trade-off relationships do exist, a notable success along one dimension may be reduced by poor performance on another.

The cognitive dimensions framework is by no means a finished entity. Meanwhile, its take-up by other researchers such as Modugno, Green and Myers (1994), Buckingham-Shum and Hammond (1994) and Yang, Burnett, DeKoven and Zloof (1995) shows the wide applicability of the approach.

6. New Directions

This final section presents several directions for further research into diagrammatic representations. They arise from our current understanding of the cognitive factors involved in use of VPLs, and build on the research described in the body of the paper. These directions were originally proposed as intentionally provocative starting points for discussion at the Thinking with Diagrams Workshop.

6.1. Basic Questions

The empirical studies that have been conducted have not yielded results in keeping with the enthusiasm of the visual programming community (Blackwell 1996b). Further success depends on the answers to these basic questions. Useful studies would shed light on what kinds of visual representations are beneficial for which tasks and especially on the appropriate class of users for a notation. We have little empirical evidence that any diagrammatic notation is responsible for improved programming performance. Three points are relevant to this issue:

First, any attempt to find such evidence should take care to separate the visual aspects of VPLs from other VPL features. New VPLs include language features orthogonal to the visual/textual dimension. Thus, a study showing a VPL outperforming a textual programming language does not necessarily mean that any visual aspect of the VPL was responsible.

Second, the question of whether a visual notation is appropriate for programming is difficult to answer since programming involves many cognitive tasks. The difficulty lies both in ascertaining which tasks account for a significant proportion of programming effort and whether a given VPL benefits those processes.

Third, the bottom line for much of the programming industry is whether a programming tool produces cost-effective results. To have a successful VPL, the issue is not whether the VPL produces a statistically significant effect, but rather whether the effect size is large enough to be of practical interest.

6.2. Metaphor and representation

Much development of visual languages is inspired by the success of the “desktop” metaphor for user interfaces. The reasons for the success of the desktop metaphor are not altogether clear, however. The principles of “direct manipulation” interfaces are agreed - manipulation of abstract entities is simplified if those entities are constrained to obey the rules of the physical world. Other than these fundamental correspondences, the value of the desktop metaphor (and especially extensions to that metaphor) have been widely questioned. Many PC users learn to use the Windows GUI without being aware of the intended metaphor, and researchers such as Halasz and Moran (1982) counsel caution in relying on metaphor.

Nevertheless, VPLs are often conceived with reference to a foundational metaphor. This metaphor generally corresponds to an underlying model of the language. Mayer (1975) demonstrated that an underlying model of the BASIC language - variables represent memory locations - was more easily learned when it was expressed in terms of metaphorical pigeon holes that represented memory locations. Blackwell (1996a) has suggested that VPLs often depend on implicit metaphors. LabVIEW, for example, is based on the data flow paradigm, and its pictorial metaphor depicts wires along which data flows.

Other psychology of programming researchers are working on the evaluation of alternative metaphorical presentations of programs. See, for example, Ploix (1996). The question for diagrammatic reasoning research is whether these metaphors have any effect other than the instructional benefits already observed by Mayer. For example, Cox (1997) has suggested that misinterpretations of common graphical notations such as Euler's circles can be attributed to use of inappropriate metaphors. Blackwell (1998), in experimental investigations, failed to

find substantial benefits of instructional or pictorial metaphors in VPL-like diagrams. A more thorough investigation of the metaphors used in VPLs may throw further light on this issue.

6.3. The role of reusable components

Many of the commercially available VPLs are promoted as being particularly suitable for end-user programming, a goal that they supposedly achieve by using a visual syntax. However, VPLS such as LabVIEW also provide users with well-stocked libraries of reusable software components. Other examples of languages providing such libraries can be found in Hils' (1992) survey of dataflow-based VPLs.

Supplying these libraries effectively raises the semantics of the language to a higher level than traditional textual programming languages. Thus, the question is raised: How much of these languages' benefits come from semantic level as opposed to their visual aspects? Furthermore, this line of reasoning could lead one to question the importance placed on metaphor. Is the issue of designing a useful VPL really one of communicating computational metaphors or is it an issue of matching the tool to the problem domain?

6.4. Programming languages for children

Programming is widely regarded as a skill that would have great educational value when taught to children. Papert's (1980) Logo language was designed for use by children, and has been used successfully for many years. Logo operates in a graphical application domain, but the language itself is purely textual. Di Sessa's Boxer (1986) was an attempt to make Logo more diagrammatic, but Boxer concentrated on issues of context and environment within which text could be organized. Two recent languages are particularly well suited to simple games and simulations. AgentSheets (Repenning & Sumner 1995) and Cocoa (originally described as "KidSim" in Smith, Cypher & Spohrer 1994) allow children to manipulate graphical elements directly, and define their behaviour within a graphical grid in terms of graphical rewrite rules and condition-action rules.

A number of commercial visual programming products have used the presentational techniques of video games to provide languages that are apparently far more attractive to children. In the mid-1980s the "ChipWits" game allowed the definition of robot behaviour by wiring logical elements together. These robots could then be placed in a maze where they competed with other robots. More recently, "Widget Workshop" provided the basic elements of a data flow language in which fanciful animated devices could be wired together – this program depiction is entertaining in its own right. In Kahn's ToonTalk™ product (Kahn 1996), three dimensional cartoon characters assemble functional programs as observable actions carried out by robots inside separate houses.

These developments are undeniably entertaining. A critical question is: are they educational? Kahn reports that 8-year old boys using ToonTalk™ take most pleasure in the fact that, once a function has completed, the house where it was evaluated explodes. This is unsurprising to anyone who has worked with children, but perhaps it reflects a deeper problem with the use of any type of diagrams by children. DeLoache & Marzolf (1992) report the difficulty children have in interpreting symbolic references. They found that more highly salient representations are far less likely to be dereferenced successfully, and that the value of the representation can actually be decreased if it is made more salient by allowing children to play with it.

The use of videogame-like presentation to teach programming to children is certain to become more popular, for the same reasons that VPLs are often immediately appealing to adults. The benefits and dangers of this trend fall within the remit of psychology of programming research, and of diagrammatic reasoning research, but they have not yet (as far as we know) been critically evaluated in these terms. This would seem to be a high priority before this type of product becomes widespread in educational settings.

6.5. Paradigm and comprehension

First generation VPLs (in the 1980s) expressed the paradigm of control flow - the order in which the program is executed - as shown in flow charts. In contrast, many current languages (including LabVIEW, VEE and Prograph) are based on the paradigm of data flow. Much empirical work on VPLs has compared relative performance between text and graphics, with little or no consideration given to how the underlying paradigm might affect comprehension. These studies may therefore be confounding the issues of diagram usage and paradigm, at least with respect to program comprehension (program design is considered in the next section). Work on this question is ongoing, and is investigating the relationship between paradigm and representation from various angles in order to assess the relative contributions of each. Control flow and data flow paradigms within VPLs have been studied by Good (1999), while Whitley (1997b) is investigating the data flow paradigm as embodied in both textual and visual notations. Navarro-Prieto (1998) has shown that the data flow organisation of spreadsheets encourages a data-structured mental model of a program.

Initial results therefore show that paradigm does make a difference, at least for novice programmer comprehension activities. Control flow VPLs seem to encourage better understanding of low-level program constructs such as program operations, actions and control flow, while data flow VPLs lead to a higher-level understanding of the program's data flow and functional aspects (Good 1999). This work casts some doubt on the common assumption that visual programming representations are mainly valuable insofar as they depict data flow. Further results which can shed more light on this issue are eagerly awaited.

6.6. Paradigm and design

Although paradigm is obviously an important factor in a programming language, studies of program design by experts suggest that design activities are not completely constrained by paradigm. Experienced programmers often plan their solution to a programming problem before they start writing any code. Studies of expert programmers show that they do not solve problems in the target programming language, but rather construct their solution strategies in a personal pseudo-language that is subsequently translated into code. (Petre & Winder 1988) This personal design representation can include pictorial mental images, whether or not the target language is a VPL (Petre & Blackwell 1997).

There is not a necessary correlation between programming languages and solution strategies; on the contrary, strategies volunteered as typical of one paradigm can often be implemented in a language that fits within another. Scholtz & Wiedenbeck (1992) observed that experts learning a new language used strategies already familiar to them, rather than learning how the features of the new language might require a different approach. Furthermore expert programmers do not observe language or paradigm boundaries when constructing solutions but rather borrow useful features across languages or domains. A paradigm might influence strategy, but it may not be the paradigm embodied in the language (Petre 1996). This implies that the cost of reasoning about a given representation might vary, depending on how the programmer's reasoning shifts. Researchers therefore need to consider reasoning paradigm separately from representation paradigm when investigating design and/or coding activities.

6.7. Multiple Representations

What are the benefits (if any) of using multiple representations in programming? Shneiderman et.al. (1977) suggested that if one of the representations is well-known and understood, then additional ones will be redundant. However, this may not necessarily be true, given that:

- different representations will highlight different types of information at the expense of others.
- understanding of a notation is often not complete and/or correct (e.g. when learning a new language, or novel applications of a language).

We know of no empirical work in the field which has addressed the use of multiple representations in program coding. However, Petre et al. (1998) have considered a number of scenarios in which multiple representations might be used in software visualisation, and many of these are applicable to coding: again, empirical confirmation of these hypotheses would be welcome.

6.8. Scalability

The scalability question (Burnett et al. 1995) asks whether a software technique suitable for solving small problems can successfully be extended to large-scale ones. This applies to many aspects of computer science. For example, a configuration suitable for a small network may become prohibitively expensive due to traffic bottlenecks as its size increases. Despite common criticisms regarding scalability, Burnett et al. claim that VPLs can successfully be applied to large problems. There are, however, few empirical studies investigating scaling issues that are specific to visual notations, such as problems of perceptual discrimination between large numbers of similar symbols.

On the other hand, there may be no demand for more scalable VPLs: Nardi (1993) has predicted that the most successful VPLs will be the ones aimed at end user programming. Thus, the real impact of these VPLs may stem from enabling more people to get more of their relatively smaller jobs done.

6.9. Software lifecycle

New programming languages are often developed with an emphasis on the processes of coding and comprehension. In practice, however, commercial software development projects spend far more effort on activities such as system specification, documentation, testing and maintenance. Whitley & Blackwell (1997)

found that commercial users of LabVIEW are concerned with these issues, and that this concern focuses on practical questions that are seldom considered in visual language research - the problem of how visual languages should be printed out, for example, or how source code control systems can construct visual delta files showing the changes between different program versions.

Investigation of the rest of the software lifecycle would be a particularly rewarding area for VPL research. It is also particularly challenging, because it requires long-term observational studies, and is not amenable to investigation through controlled experiments. As our review of empirical studies shows, even small-scale controlled studies have found little conclusive evidence regarding the benefits of VPLs. Large-scale studies of real development projects are unlikely to produce substantially more clearcut results.

7. Conclusions

This paper has described the origins and characteristics of the class of diagram described as visual programming languages, and has reviewed techniques which have been used to study these diagrams within the context of psychology of programming.

A number of researchers have carried out empirical studies using these techniques to directly compare problem solving performance using diagrams and textual notations. These studies have not been conclusive as was hoped, but we do not wish to be overly critical of the intuitive position regarding the value of visual representations. The challenge to researchers in thinking with diagrams is to explain why this intuition might be valid, and to propose the ways in which, if it is valid, it can most effectively be exploited.

The empirical studies completed to date have, however, formed a stepping stone for broader frameworks of notational analysis such as Green's Cognitive Dimensions. These new integrative approaches are moving away from simplistic comparisons of text and graphics, in order to investigate the ways in which tasks can be matched to appropriate representations.

Visual programming is a particularly fertile area for studying the application of diagrams to real-world problem solving contexts. This paper has proposed a number of research questions that are of particular relevance at the time of writing. Any of these would be both challenging and valuable as a starting point for future diagrammatic reasoning research.

9. References

Baroth, E. & Hartsough, C. (1995). Visual programming in the real world. In Burnett, M., Goldberg, A. & Lewis, T. (Eds), Visual Object-Oriented Programming: Concepts and Environments, chapter 2, pages 21-42. Manning Publications Co.

Blackwell, A.F. (1996a). Metaphor or Analogy: How Should We See Programming Abstractions? In P. Vanneste, K. Bertels, B. De Decker & J.-M. Jaques (Eds.), Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group, pp. 105-113.

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.

Blackwell, A.F. (1998). Metaphor in diagrams. Unpublished PhD thesis. University of Cambridge.

Buckingham-Shum, S. & Hammond, N. (1994) Argumentation-based design rationale: what use at what cost? International Journal of Human-Computer Studies, **40**, 603-632.

Burnett, M.M., Baker, M.J., Bohus, C., Carlson, P., Yang, S. and van Zee, P. (1995). Scaling Up Visual Programming Languages. IEEE Computer **28**(3):45-54.

Cox, R. (1997). Representation interpretation versus representation construction: An ILE-based study using SwitchERII. In Proceedings of the 8th World Conference of the Artificial Intelligence in Education Society (AI-ED 97), Amsterdam: IOS Press, 'Frontiers in Artificial Intelligence and Applications' series.

Cunniff, N. & Taylor, R.P. (1987). Graphical vs. textual representation: An empirical study of novices' program comprehension. In Empirical Studies of Programmers: Second Workshop, pp. 114-131.

Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J. & Boehm-Davis, D.A. (1989). Experimental evaluation of software documentation formats. Journal of Systems and Software, **9**(2), 167-207.

DeLoache, J.S. & Marzolf, D.P. (1992). When a picture is not worth a thousand words: Young children's understanding of pictures and models. Cognitive Development, **7**, 317-329

diSessa, A. (1986). Notes on the future of programming: breaking the utility barrier. In D.A. Norman & S.W. Draper (Eds), User Centered System Design: New Perspectives on Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum.

Ferguson, E.S. (1992). Engineering and the Mind's Eye. Cambridge, MA: MIT Press.

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

Gilmore, D.J. & Green, T.R.G. (1984). Comprehension and recall of miniature programs. International Journal of Man-Machine Studies, **21**, 31-48.

Glinert, E.P. (Ed.). (1990a). Visual programming environments: Applications and issues. IEEE Computer Society Press.

Glinert, E.P. (Ed.). (1990b). Visual programming environments: Paradigms and Systems. IEEE Computer Society Press.
Good, J. (1999). Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations into the Factors involved in Novice Program Comprehension. Unpublished PhD thesis. University of Edinburgh, Department of Artificial Intelligence.
Green, T.R.G. (1977). Conditional programs statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, **50**, 93-109.
Green, T.R.G. (1989) Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), People and Computers V. Cambridge: Cambridge University Press. pp. 443-460.
Green, T.R.G. (1991) Describing information artefacts with cognitive dimensions and structure maps. In D. Diaper and N.V. Hammond (Eds.), People and Computers VI. Cambridge: Cambridge University Press. pp. 297-315.
Green, T.R.G. & Petre, M. (1992). When visual programs are harder to read than textual programs. In Proceedings of Sixth European Conference on Cognitive Ergonomics (ECCE6), pp. 167-180.
Green, T.R.G. & Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing, **7**, 131-174.
Green, T.R.G., Petre, M. & Bellamy, R.K.E (1991). Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In Empirical Studies of Programmers: Fourth Workshop, pp. 121-146.
Gurka, J.S. & Citrin, W. (1996). Testing effectiveness of algorithm animation. In Proceedings of 1996 IEEE Symposium on Visual Languages (VL'96), pp. 182-189.
Halasz, F.G. & Moran, T.P. (1982) "Analogy considered harmful" Proc. Conf. Human Factors in Computing Systems: ACM
Hils, D.D. (1992) Visual Languages and Computing Survey: Data Flow Visual Programming Languages. Journal of Visual Languages and Computing, **3**, 67-101.
Hoc, J.-M., Green, T.R.G., Samurçay, R. & Gilmore, D.J. (1990). Psychology of Programming. London: Academic Press.
Kahn, K. (1996) ToonTalk™ – An animated programming environment for children. Journal of Visual Languages and Computing, **7(2)**, 197-218.
Mayer, R.E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, **67**(6), 725-734.
Modugno, F., Green, T.R.G., Myers, B.A. (1994) Visual programming in a visual domain: a case study of cognitive dimensions. In: People and Computers IX. Cambridge University Press. Cambridge. 91-108.
Moher, T.G., Mak, D.C, Blumenthal, B. & Leventhal, L.M (1993). Comparing the comprehensibility of textual and graphical programs: The case of petri nets. In Empirical Studies of Programmers: Fifth Workshop, pp. 137-161.
Myers, B. A. (1990). Taxonomies of visual programming and program visualization. Journal of Visual Languages and Computing, **1**, 97-123.
Nardi, B.A. (1993) A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA: MIT Press.
Navarro-Prieto, R. (1998). The role of imagery in program comprehension: Visual programming languages. Doctoral Thesis, University of Granada. ISBN 84-8497-957-1.
Newell, A. & Simon, H.A. (1972). Human Problem Solving. Englewood Cliffs, NJ: Prentice Hall.
Pandey, R.K. & Burnett, M.M. (1993). Is it easier to write matrix manipulation programs visually or textually? An empirical study. In Proceedings of 1993 IEEE Symposium on Visual Languages (VL'93), pp. 344-35.
Papert, S. (1980). Mindstorms, Children, Computers and Powerful Ideas. New York: Basic Books.
Petre, M. (1991) Shifts in reasoning about software and hardware systems: must operational models underpin declarative ones? In: The Third Workshop of the Psychology of Programming Interest Group (Hatfield, January).
Petre, M. (1995). Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM, 38(6), 33-44.
Petre, M. (1996) Programming paradigms and culture: implications of expert practice. In: M. Woodman (Ed.) Programming Language Choice: Practice and Experience. London: International Thomson Computer Press. 29-44.
Petre, M. & Blackwell, A.F. (1997). A glimpse of expert programmer's mental imagery. In S. Wiedenbeck & J. Scholtz (Eds.), Proceedings of the 7th Workshop on Empirical Studies of Programmers, pp. 109-123.
Petre, M., Blackwell, A.F., & Green, T.R.G. (1998). Cognitive Questions in Software Visualisation. In J. Stasko, J. Domingue, M. Brown & B. Price (Eds.) Software Visualization: Programming as a Multi-Media Experience. Cambridge, MA: MIT Press. pp. 453-480
Petre, M. & Green, T.R.G. (1993). Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. Journal of Visual Languages and Computing, **4**, 55-70.
Petre, M. & Winder, R. (1988) Issues governing the suitability of programming languages for programming tasks. In: People and Computers IV. Cambridge University Press.
Ploix, D. (1996). Building program metaphors. In M. Ireland (Ed.), Proceedings of the First Psychology of Programming Interest Group Postgraduate Student Workshop. pp. 125-129.
Price, B.A., Baecker, R.M. & Small, I.S. (1993). A principled taxonomy of software visualization. Journal of Visual Languages and Computing, **4**(3), 211-266
Reitman, W.R. (1964). Heuristic design procedures, open constraints and the structure of ill-defined problems. In M.W. Shelly & G.L. Bryan (Eds.) Human Judgements and Optimality. New York: Wiley and Sons.
Repenning, A. & Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented visual languages. IEEE Computer, 28(3), 17-25.
Scanlan, D.A. (1989). Structured flowcharts outperform pseudocode: An experimental comparison. IEEE Software, **6**(5), 28-36.
Shneiderman, B. (1980). Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop.

Shneiderman, B., Mayer, R., McKay, D. and Heller, P. (1977) Experimental Investigations of the Utility of Detailed Flowchart s in Programming. Communications of the ACM, **20**, 373-381.

Smith, D.C., Cypher, A. & Spohrer, J. (1994). KIDSIM: Programming agents without a programming language. Communications of the ACM, **37**(7), 55-67.

Stasko, J.T., Badre, A.M. & Lewis, C. (1993). Do algorithm animations assist learning? An empirical study and analysis. In Proceedings of INTERCHI '93 Conference on Human Factors in Computing Systems, pages 61-66.

Vessey, I. & Weber, R. (1986). Structured tools and conditional logic: An empirical investigation. Communications of the ACM, **29**(1), 48-57.

Weinberg, G.M. (1971). The Psychology of Computer Programming. New York: Van Nostrand Reinhold.

Whitley, K. (1997a). Visual programming languages and the empirical evidence for and against. Journal of Visual Languages and Computing, **8**(1), 9-142.

Whitley, K. (1997b). In search of visual programming. In C. Kann (Ed.), Proceedings of the First Empirical Studies of Programmers Graduate Student Workshop, October 24, 1997.

Whitley, K.N. & Blackwell, A.F. (1997). Visual programming: The outlook from academia and industry. In S. Wiedenbeck & J. Scholtz (Eds.), Proceedings of the 7th Workshop on Empirical Studies of Programmers, pp. 180-208.

Yang, S., Burnett, M.M., DeKoven, E., & Zloof, M. (1995) Representation design benchmarks: a design-time aid for VPL navigable static representations. D.C.S. Tech Rpts 95-60-3. Oregon State University (Corvallis).