USABILITY ISSUES AND DESIGN PRINCIPLES FOR VISUAL PROGRAMMING LANGUAGES

A thesis submitted for the degree of Doctor of Philosophy

by

Jarinee Chattratichart

Department of Information Systems and Computing

Brunel University

2003

Abstract

Despite two decades of empirical studies focusing on programmers and the problems with programming, usability of textual programming languages is still hard to achieve. Its younger relation, visual programming languages (VPLs) also share the same problem of poor usability.

This research explores and investigates the usability issues relating to VPLs in order to suggest a set of design principles that emphasise usability. The approach adopted focuses on issues arising from the interaction and communication between the human (programmers), the computer (user interface), and the program. Being exploratory in nature, this PhD reviews the literature as a starting point for stimulating and developing research questions and hypotheses that experimental studies were conducted to investigate. However, the literature alone cannot provide a fully comprehensive list of possible usability problems in VPLs so that design principles can be confidently recommended. A commercial VPL was, therefore, holistically evaluated and a comprehensive list of usability problems was obtained from the research. Six empirical studies employing both quantitative and qualitative methodology were undertaken as dictated by the nature of the research. Five of these were controlled experiments and one was qualitative-naturalistic.

The experiments studied the effect of a programming paradigm and of representation of program flow on novices' performances. The results indicated superiority of control-flow programs in relation to data-flow programs; a control-flow preference among novices; and in addition that directional representation does not affect performance while traversal direction does – due to cognitive demands imposed upon programmers. Results of the qualitative study included a list of 145 usability problems and these were further categorised into ten problem areas. These findings were integrated with other analytical work based upon the review of the literature in a structured fashion to form a checklist and a set of design principles for VPLs that are empirically grounded and evaluated against existing research in the literature. Furthermore, an extended framework for Cognitive Dimensions of Notations is also discussed and proposed as an evaluation method for diagrammatic VPLs on the basis of the qualitative study.

The above consists of the major findings and deliverables of this research. Nevertheless, there are several other findings identified on the basis of the substantial amount of data obtained in the series of experiments carried out, which have made a novel contribution to knowledge in the fields of Human-Computer Interaction, Psychology of Programming, and Visual Programming Languages.

Acknowledgements

The journey to completing a PhD has *not* been a lonely path, as I was told when I started. Of course, there were times when I experienced intellectual loneliness, which was, perhaps, the feeling I most resented. But the world is not static, time passes, and new people come around. Only through this long journey have I had the opportunity to befriend and to be inspired by so many people from different professions, cultures, and expertise. They are my friends who, one way or another—intellectually or not—have been a part of this research and a part of the new *me* at the end of this journey.

Firstly, I thank my long-term teacher and mentor, Dave Sharp, for his faith in me and for always being supportive. I am indebted to my supervisors, Dr Jasna Kuljis and Professor Ray Paul, for their supervision and, in particular, for their patience in waiting for and reading the 'Story'. I specially thank all participants of my experiments at the University of Westminster, University of London (Goldsmiths campus), and Brunel University.

In the early days of this research, my thanks go to Professor Thomas Green, who joyfully answered my questions both via emails and over dining tables. At Brunel, I thank Jacqueline Brodie and Chiladda Chennawasin, my PhD buddies who, frequently came to the rescue of my 999 'intellectual' calls. Those who inspired and helped me over the Atlantic via emails must also be acknowledged. I thank Professor Gitte Lindgaard of Carleton University, Canada. My mentor and friend, through emails at low and high times and, without whom, HE-Plus may not have been born. I would also like to thank Dr Paul D. Tynan at Eastman Kodak in New York for sharing with me his professional experience. I was fortunate enough to gain confidence in a piece of my work in this research by having known Dr Patrick W. Jordan who shared my faith in the Immersion method.

The list only gets longer and there seems not to be enough space to mention all who must be thanked. On the personal front, I thank Mano, Mina, Kana, and Seiji for putting up with my bad habit of working '*anytime and anywhere*'. Perhaps, they did not have a choice, but I am still grateful. There were others who helped shape the new *me* during this journey. Jacqueline, who arrived during my second year at Brunel kept me sane by distracting me with the fun of researching our shared interests. This distraction was an invaluable experience such that I gladly paid the price of a longer journey towards completing my PhD. Jacqueline, however, was not the only supporter of my mental well-being, I also owe my sanity and fitness to Nick Adamou and his karate training, which I would have never tried had it not been for the monotonic lifestyle that I adopted during this journey.

Finally, I thank you all for your serious effort in proof reading this thesis: Jacqueline Brodie, James Evans, Robert Scane, and Nick West.

List of Publications

2003

1. Brodie, J., Chattratichart, J., Perry, M., & Scane, R. (2003). How age can inform the future design of mobile phone experience. In S. Constantine (Ed.), *Proceedings of the Human Computer Interaction International, HCII 2003*, Volume 4 (pp.822-826). NJ:Lawrence Erlbaum Associates.

2. Chattratichart, J. (2003). Establishing Design Principles for Diagrammatic VPLs. In M. Rauterberg, M. Menozzi & J. Wesson (Eds.), *Proceedings of the Ninth IFIP TC.13 Conference on Human-Computer Interaction, Interact 2003*(pp. 948). Amsterdam: IOS Press.

3. Chattratichart, J. & Brodie, J. (2003a). HE-Plus: Toward usage-centered expert review for website design. In L. L. Constantine (Ed.), *Proceedings offorUSE 2003, Second International Conference on Usage-Centered Design* (pp. 155-169). Massachusetts: Ampersand Press.

4. Chattratichart, J. & Brodie, J. (2003b). Envisioning a mobile phone for 'all' ages. In M. Rauterberg, M. Menozzi & J. Wesson (Eds.), *Proceedings of the Ninth IFIP TC.13 Conference on Human-Computer Interaction, Interact 2003* (pp. 725-728). Amsterdam: IOS Press.

5. Chattratichart, J. & Brodie, J. (2003c). Inclusive phone design to bridge the age gap. In *Proceedings of the Fourth Annual ACM SIGCHI-NZ Conference on Computer-Human Interaction, CHINZ 2003* (pp.111-115). ACM SIGCHI.

6. Chattratichart, J. & Brodie, J. (2003d). The age factor in the design equation of cell phones. In *Proceedings of the 12th Annual Usability Professionals' Association Conference, UPA 2003*. UPA.

7. Chattratichart, J., Cave, D. & Vaduva, A. (2003). Learning and doing 'expert evaluation': A teaching dilemma. In L. L. Constantine (Ed.), *Proceedings offorUSE 2003, Second International Conference on Usage-Centered Design* (pp. 27-35). Massachusetts: Ampersand Press.

8. Chattratichart, J. & Jordan, P. W. (2003). Simulating 'lived' user experience – Virtual immersion and inclusive design. In M. Rauterberg, M. Menozzi & J. Wesson (Eds.), *Proceedings of the Ninth IFIP TC.13 Conference on Human-Computer Interaction, Interact 2003* (pp. 721-725). Amsterdam: IOS Press.

9. Chattratichart, J., Turner, C. & Brodie, J. (2003). Exploring the total customer experience: Usability evaluations of (B2C) e-commerce environments. In S. Minocha & L. Dawson (Eds.), *Proceedings of Workshop 6, the Ninth IFIP TC.13 Conference on Human-Computer Interaction, Interact 2003* (pp. 4-5). The Open University.

10. Jordan, P. W. & Chattratichart, J. (2003). Immersion and design – Getting inside the user's mind. In *Proceedings of the ICSID 2nd Educational Conference, ICSID Design Congress 2003* (pp.48-52). IF International Forum Design GmbH.

2002

11. Brodie, J. & Chattratichart, J. (2002). Contextualising heuristic evaluation to improve website appraisal. In *Proceedings of the 11th Annual Usability Professionals' Association Conference, UPA 2002* (pp. 51). UPA.

12. Chattratichart, J. & Brodie, J. (2002a). Establishing design guidelines for a better online shopping experience. In *Proceedings of the 11th Annual Usability Professionals' Association Conference, UPA 2002* (pp. 51). UPA.

13. Chattratichart, J. & Brodie, J. (2002b). Extending the heuristic evaluation method through contextualisation. In *Proceedings of the 46th Annual Meeting of the Human Factors and Ergonomics Society, HFES 2002* (pp. 641-645). HFES.

    [This paper was nominated for the Alphonse Chapani's Best Student Paper Award at HFES 2002.]

14. Chattratichart, J. & Kuljis, J. (2002). Exploring the effect of control-flow and traversal direction on VPL usability for novices. *Journal of Visual Languages & Computing, 13*(5), London: Academic Press, 471-500.

2001

15. Chattratichart, J. & Kuljis, J. (2001a). Some evidence for graphical readership, paradigm preference, and the match-mismatch conjecture in graphical programs. In G. Kadoda (Ed.), *Proceedings of the 13th Annual Meeting of the Psychology of Programming Interest Group, PPIG 2001* (pp. 173-189). Sheffield: Print Unit.

16. Chattratichart, J. & Kuljis, J. (2001b). Why diagrams are sometimes difficult. In H. Michitaka (Ed.), *Proceedings of the Eighth IFIP TC.13 Conference on Human-Computer Interaction, Interact 2001,* Volume 2 (pp.755-756).

2000

17. Chattratichart, J. (2000). Visualisation of program specification. In S. Turner and P. Turner (Eds.), *Proceedings of Human Computer Interaction, HCI 2000* (Vol.2, pp. 145-146).

18. Chattratichart, J. & Kuljis, J. (2000a). An assessment of visual representations for the 'flow of control'. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group, PPIG 2000* (pp. 45-58). Cosenza, Italy: Memoria.

19. Chattratichart, J. & Kuljis, J. (2000b). A comprehensibility comparison of three visual representations and a textual program in two paradigms. In *Proceedings of the Visual End User Workshop* (pp. 104-119).

Table of Contents

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Problem Explanation

Visual programming languages (VPLs) let programmers specify programs graphically or visually. The claim that programming visually is an easier process than textual programming has been one of the major motivations for research in the VPL community. For example, Myers (1990) claims that human visual information processing systems are optimised for multi-dimensional data. Similarly, Scanlan (1989) states that graphical programs require the use of both left and right hemispheres of the brain simultaneously to process both logic and graphics. Shu (1992) maintains that pictures are more powerful than words, aid understanding and remembering, provide an incentive to learning to program, and do not impose language barriers. However, Blackwell (1996) demonstrates that these are merely metacognitive beliefs (beliefs that one has about the way one carries out mental tasks), some of which are founded but others are not. Blackwell (1996) and Whitley (1997), thus call for more empirical evidence to support these claims. A decade has elapsed since the VPL boom in the 1990s but VPLs are still not widely used. Why should this be the case?

Early visual programming systems and languages were developed and designed for specific purposes such as teaching programming students. Examples of these are FPL or First Programming Language (Taylor *et al.*, 1986), BridgeTalk (Bonar & Liffick, 1990), and Pursuit (Modugno & Myers, 1994). These languages were domain specific, limited in functionality, and, despite the claim that they have either been designed using approaches based on empirical research or to help improve student programmers' performance, still remain as prototypes. The rate that VPLs have penetrated the programming language market is slow. Today, only a few commercial VPLs are available and only one is a truly general-purpose program language – Prograph VPL (Blackwell *et al.*, 2001). Furthermore, none of these commercial VPLs are used as a teaching programming language. Perhaps, then, merely being 'visual' does not warrant pre-supposing VPLs easy to use and to learn.

Indeed, there has been some empirical evidence to suggest that programs written in two widely used commercial VPLs (LabVIEW and Prograph) are not easier to understand than those written in textual languages (Green *et al.*, 1991; Green & Petre, 1996). In one study, Green, *et al.* (1991) provided evidence that the LabVIEW program tested in their study was inferior to its equivalent textual program. In another study, Green & Petre (1996) conducted a straw comparison between three equivalent programs written in Prograph, LabVIEW, and Basic. They found that the two VPLs performed extremely poorly. Using the Cognitive Dimensions of Notations framework (Green, 1989) as an inspection method, they also evaluated the usability of these two VPLs on the same occasion. The results of their evaluations showed that the aspects concerning Human-Computer Interaction of these two visual programming languages were still "underdeveloped" (Green & Petre, 1996).

1.2 Statement of research objective and its scope

The main objective of this research is to investigate and attempt to identify usability problems surrounding VPLs in order to produce a checklist and design principles for VPLs that emphasise usability. Since most successful commercial VPLs (e.g. LabVIEW and Prograph) are of a diagrammatic type, the scope of this research is limited to investigating usability issues of diagrammatic VPLs so that its findings can readily benefit the present VPL community. Furthermore, the investigation and empirical studies carried out focused on novices. This is because some of the severer problems encountered by novices may be too subtle to be detected by expert programmers.

In order to make this research manageable, this work is limited to the issues of interactivity between the program and the programmer and does not delve too deeply into diagram reasoning.

1.3 Terms and definitions

Notation

The term *notation* used in this thesis refers to a programming language or a system of diagrammatic representations.

Perceptual coding of programs

In this thesis this term refers to the combination of visual elements or attributes in the program or programming environment that conveys an intended meaning (accurately or not) of the programmer to readers (himself or others), helps or hinders readers' ability to recognise the existence of, to understand the meaning of, or to differentiate between, different visual objects used in the program. Examples of visual elements are icons, buttons, windows, white space, layout, colour, shadow, thickness, highlight, font type and style, etc. Perceptual coding in our definition is different fron the term '*secondary notation*' defined by language (Green & Petre, 1996) as refering to code that are used as an extra means to improve the program beyond the 'official' semantics of the programming.

Usability

There are many definitions of "usability" defined by different standards and authors. The definition given by the ISO/IEC 9126-1 standard for Software Product Quality Model is adopted here. The reasons are, firstly, its definition agrees with those of other authors, such as Nielsen (1993) and Shackle (1991) and, secondly, this definition excludes functionality (Bevan, 2002), which helps limit the scope of this thesis. The ISO/IEC 9126-1 (Bevan, 2002) defines *usability* as "the capacity of the software product to be understood, learned, used and attractive to the user, when used under specified conditions". It must be noted that this definition concerns the product's understandability, learnability, operability, and attractiveness. The former three qualities are somewhat related. However, attractiveness is more concerned with pleasure, feeling, and emotion. It requires investigation into the studies of pleasure-based approach to human factors (Jordan, 2000), hence making the scope of this research much wider than the time frame of this research would allow. Therefore, attractiveness is not included in the definition used in this thesis.

In summary, "usability" in this thesis refers to understandability, learnability, and operability. It is the capacity of the product to be understood, learned, and used under specified conditions.

1.4 Approaches to the problem

Poor usability is a problem not limited to VPLs but includes textual programming languages as well. In an informal poll carried out on the Web in 2001 by Kuro5hin, a technical and culture organisation ("Programming languages have the usability of a", n.d.), respondents were asked to identify an object, from a given list, whose usability matched that of a programming language. They had to choose from a toaster, power point, model T Ford, Boeing 767, spoon, catapult, automatic hand dryer, or web page. The result showed that the highest vote was for Boeing 767 (32%) and the lowest vote was for a toaster (4%) and an automatic hand dryer (4%). As non-academic as this trivial poll may be, its result from the votes of this technological-minded Internet user group, does give a clear message of the perceived poor usability of programming languages.

As the lessons learned from textual programming languages community may well be useful to the relatively young VPL community, it is therefore sensible to look at what approaches have been or can be adopted to make programming languages easier to use or to learn.

1.4.1 Improving the programming environment

One way to make the programming process faster and easier is to improve the programming environment by providing a good program editor, on-line help and debugging facilities, animation, visualising facilities, and so forth. However, programming language software often provides far too many features which are rarely used and are particularly useful only to experts but are confusing to novices. A programming environment requires programmers to perform non-programming related tasks in addition to writing and debugging programs. Therefore, novices must learn both how to program and how to work effectively in the environment. The environment should thus be made as simple to use as possible. It should not impose any obstacle to, but possibly help ease, the programming process. Nonetheless, improving the programming environment does not directly address all the problems for programmers, if the language itself is difficult to learn and use for novices.

1.4.2 Instincts—heuristics—functionality—speed

Later releases of programming languages tend just to be modified versions of previous releases to fix problems encountered in earlier versions. Experiences gained and lessons learned from the problems of old or existing languages are valuable for future designs. That is, designers can use their prior experience and instincts and apply rules of thumb and heuristics in design. However, it is not easy to anticipate all the programmers' needs and preferences. Therefore, it is not uncommon to see more functions and features than would seem necessary or many features that allow programmers to do the same task. This approach seems sensible and is as good as one can get provided the complexity of the programming language applications.

Given the nature of their complexity (despite empirical research being conducted for over two decades) empirical studies of programming languages tend to be narrowly focused on a small subset of features or functions of interest. A published set of research-based design principles for programming languages is hard to find, let alone finding any standard for language design. In fact, there has been only one summarised by Myers (n.d.), based on Nielsen's (1993) heuristic evaluation method. The principles were drawn from examples in C, C++, Java, PERL, Visual Basic, and HyperCard. However, these principles are not empirically grounded because heuristic evaluation is an inspection method and, hence, predictive.

Lacking a well-established set of design principles, language designers have thus been left to relying on their own instincts, experience, and rule-of-thumb heuristics. This approach has its own problem. What the designers think to be obvious or easy may not be the case with programmers – experts or novices alike. Prior experience and heuristics, followed by generations of programming language design, could be useful but to what extent and in what context is an open question.

Another practice is to alleviate the novices' frustration during programming by providing more programming language functionality and improving on program execution efficiency. However, this is not a solution to poor usability problem of programming languages. If a language is difficult to use and to learn for novices, it will still take them a long time to successfully debug their programs. Furthermore, rather than helping the programming language to be simpler and easier to learn or use, some added functionality, such as having many ways to do the same thing, could make learners of the programming language more confused.

In short, added functionality, improving on program execution speed, designer's instincts and existing rules of thumbs without empirical support, we can argue, may not be the sole answer to designing an easy to use language after all.

1.4.3 Focusing on the *human* in design

Programming languages are used by humans to instruct machines how to solve particular problems. They should therefore be designed with an emphasis on maximising human performance while compromising machine and implementation efficiency. There is no need to have a language that gives high machine performance but low programmer performance, which in turn increases human resource requirements; or vice versa. The programmers themselves are central to this approach. Vessey & Weber (1986) once stated that (textual) programming languages should be "designed with an understanding of psychological processes that programmers must bring to bear on a task" or "with an understanding of the representation that best facilitates the task to be performed". VPL designers should do the same. In designing a new language it is important to consider psychological processes that take place during programming and to consider the interaction between programmers and the programs. Findings from empirical studies of programmers could provide the designers with some insights into problems with programming. Identifying what aspects of programming languages make programming hard for novices can help guide the new design. However, considering the relationship between the programmers and the programming languages alone is inadequate. Today the programming tasks are usually carried out on a computer. Programmers do not write a program on paper and pen any more. Interactions between the programmers, the computer and user interface issues should also be taken into account by language designers. Particularly, for visual programming, the programmer's interactions with visual representations of the program may not be a trivial matter because different representations for the same programming construct may have different effects on the programmer performing the same programming tasks. As an example, here is an excerpt from an online discussion group (Scrymarch, 2001): "When programmers, the most expert users, are confronted with a new expert interface, you get interface rage to the power of ten".

In brief, then, this approach exploits the knowledge in Psychology of Programming and in HCI. This approach is not new. Manufacturers such as Apple, Sun, IBM, and Microsoft, to say the least, all have their user interface laboratories to carry out usability testing (see for example, "Sun usability labs and services", n.d., "We have over 25 labs", n.d.). However, this approach is not quickly or easily achieved. For example, IBM has had this practice, i.e. carrying out usability testing on programming language functions for more than two decades (personal communications with Dr Paul D. Tynan, a former IBM usability engineer for 17 years) and Microsoft Usability Group has been in place since 1988 ("What is the Microsoft Usability Group all about", n.d.). However long and winding the road towards usability for programming languages seems to be, this path is worth following.

1.5 Research Context

This research examines usability issues of programming languages that can inform the design of a visual programming language. However, to truly understand these issues we must investigate research from various fields, in particular, those involving understanding the interaction between the human, the computer and psychological issues relating to programming itself. The following sections provide the reader with a brief background into the various fields that form a foundation to this research.

1.5.1 VPLs in brief

Shu (1992) defines a visual programming language as "a language which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language". VPLs, in particular, diagrammatic languages have their origins in graphical programming. Graphical programming refers to programming that uses graphical representations of programming constructs as well as program flow. Graphical programs are specified using some forms of diagrams such as flowcharts and structured flowcharts. During the flowchart era in 1980s, these diagrams were used as program documentation tools. Gradually they found their places in some interactive systems as static diagrams used to aid programming (Reiss, 1984) or as executable diagrams (Pong & Ng, 1983; Frei *et al.*, 1978; Taylor *et al.*, 1986; Albizuri-Romero, 1984).

The end of the 1980s saw a rapid advance in hardware technology making implementation of graphics faster and cheaper. Graphical representations of programs were no longer restricted to diagrams consisting of geometric shapes. Direct manipulation and various kinds of visual representations such as icons, images, and graphical shapes became much easier to implement. It became economically viable to develop graphical programming systems and the term visual programming was thus coined.

The 1990s marked the beginning of a new era in visual programming research and many varieties of systems and languages were implemented. Burnett & Baker (1994) classify visual representations used by VPLs into three types: diagrammatic, iconic, and static pictorial. This research focuses on diagrammatic VPLs.

1.5.2 Interactions between human and the program

There has been nearly two decades of research studying the nature of programs and programming tasks, the problems that programmers experience, programming strategies, mental models of programs, expert programmers versus novice programmers, and so on. Reviewing the literature in this area promotes understanding of the human aspects and cognitive issues of the programming process. However, much of the work in this area has been based on textual programming languages. VPLs have only become a subject of study for a handful of research projects in this field since the 1990s. Therefore, there is not much research directly relevant to our investigation into the psychological issues of interactions between programs and programmers in the literature. There is, therefore, a need for us to look at the available research studying textual programming languages - even though it is not known whether research findings from textual programming languages can be extrapolated to visual programming languages. We, inevitably, begin our investigations with a presumption that what programmers look for in a textual program should be similar to, if not exactly the same as, that in an equivalent visual program.

Since the 1970s empirical research into the psychology of programming has been conducted to study programmers' performances on various programming activities: coding, comprehension, modification, and debugging. Comprehension plays an important role in programming and will be the area that we focus on. This is because comprehension forms a common ground for all other activities. To modify or debug a program, the programmer needs to comprehend it first. Coding may not seem to require comprehension, however, it does. Programmers write programs incrementally. They tend to write a small piece of code, read it, understand it, find mistakes, modify the code, and add some more code. In other words, the process of coding consists of iterations of write-read-comprehend. Thus, comprehension is the key programming activity. If a program cannot be easily comprehended, it is of little use. We will, therefore, offer a review of the programming comprehension literature that addresses psychological issues pertaining to the interactions between the programmers and the programs, using both textual languages and VPLs.

1.5.3 Interactions between human and the computer

In conventional design methodologies, emphasis is given to system functionality and implementation rather than real users of the system. Users (usually managers level or above) are involved in the process of function specification. The finished product is not evaluated or tested against the human users who use the system in their day-to-day work. On the other hand, the user-centred design approach emphasises users' involvement throughout the design process. Central to this approach is the iterative design methodology in which the design process consists of a 'design-implement-evaluate loop'. This is because, as Gould (1995) stated: a. "Nobody can get it right the first time"; b. "Development is full of surprises"; and c. "Developing user-oriented systems requires living in a sea of changes".

The two vital elements that form the design-implement-evaluate loop are: focusing on users and user testing (see for example, Rubinstein & Hersh, 1984; Gould & Lewis, 1985; Shneiderman, 1992; Nielsen, 1993; Mayhew, 1999). In the first element, looking at users, the users are the main focus in the process of requirements capturing, which is based on user profiling and task analysis, i.e. knowing who will use the system, what their characteristics and their tasks are, and the workflow (how users carry out their tasks). Knowledge derived from this first element informs design. The second element is evaluation of the designed prototypes (formative evaluation) or the final products (summative evaluation) to check whether such a design is acceptable when the users are faced with actually doing some tasks using them. This element, namely, usability evaluation, is indispensable to ensure user's acceptance of the design.

The benefit of taking the user-centred design approach for our research is that issues or problems relating to the interactions between the computer (the user interface) and the human users (programmers) can be addressed and revealed. Our investigations will be carried out, not only by a review of the literature in the field of Human-Computer Interaction (HCI), but also by actually evaluating the usability of an existing VPL, which will reveal information that we hope will inform our checklist and principles for VPL.

1.5.4 From the program to the human: communication through visual language

A program can communicate its meaning to the programmer through its perceptual characteristics that form the 'visual language'. The term 'visual language' is not the same as 'visual programming language' in this thesis. Visual language is defined as "the tight coupling of words, images, and shapes into a unified communication unit" (Horn, 1998). It refers to the verbal and visual elements on a document (of any medium) that conveys some meanings to the reader, such as, colour, layout, symbols, white spaces, and indentation (Marcus, 1992). It is the combination and overall effect of these signs that makes up the visual language of a document (Kostelnick & Roberts, 1998). Words, images, and shape that form a piece of a visual language cannot be removed without altering the meaning of the information it originally represents (Horn, 1999). Visual language can thus be used as a means to help communicate the meaning of the program to the programmers. Two programs that do the same thing (i.e. having the same meaning) and written exactly in the same way (e.g. using same programming statements or graphical symbols) but using different combinations of visual elements (such as layout, colour, etc.) may entail different programming performances. The virtue of a visual language has already been and is increasingly recognised among information architects, graphic designers, and web designers. In programming, however, exploitation of the virtues of a visual language is quite limited to indentation, fonts, and colour (in textual programming languages) and at times can be ad hoc (in VPLs). Among the areas that we will investigate is how a visual language can be used to enhance comprehensibility of a visual program.

1.6 Research Methodology

In choosing research methods for this research, we consider the following factors:

1. The purpose of the research question

First of all, for each of our research questions we considered whether our aim was for discovery or testing some hypotheses. For the former, the research is exploratory in nature and requires in-depth analyses in interpreting field data. Data analyses are mostly qualitative. However, the latter assumes that causal relationship exists and, therefore, hypotheses are formed and tested using experimental method, for example. The data obtained are analysed, mostly, quantitatively. Therefore being able to differentiate between these two types of research questions is critical to choosing an appropriate research method.

2. Resources available.

Secondly, we matched our resources with what is called for by the research methods' potential suitability for the research questions we have. Sometimes, to be realistic, trade-offs were made in making decisions about methods. This is because the best or most ideal method may also impose a high demand on resources (number of researchers and participants in an empirical study), budget, and project duration. We use methods or combination of methods, research tools, and techniques that are most appropriate and plausible for our research questions.

To serve the objectives of this research, many research questions were asked during the course of our investigations. Some are exploratory in nature whilst others, use hypotheses. Therefore, we have employed methods belonging to both ends of the methodology continuum. These include critical research review, experimental method, and qualitative inquiry. Research review provides a theoretical background of the knowledge required to form hypotheses and/or research questions relevant to the objectives of this research but have not yet been tackled by others. The experimental method is employed for testing these hypotheses. Finally, a qualitative inquiry is employed in a holistic evaluation of a VPL by a technique called Immersion (Jordan, 2000) to explore potential usability problems/issues in VPLs.

For triangulation purposes, a combination of research methods, data collection techniques, and different statistical methods are employed. For example, whilst the experimental method generates precise measurement data, pre-test and post-hoc questionnaires are also used as other means to collect qualitative data. Whilst the qualitative inquiry generates thick descriptive qualitative data, statistical data analysis is also carried out from quantitative data derived from the narrative description.

1.7 The structure of this thesis

This chapter has introduced the problems that surround the research, laid out the context of the research, stated its objectives, and discussed the multi-disciplinary approach taken. The rest of the chapters in this thesis are organised based upon topics of investigation, many of which may seem unrelated but are, in fact, relevant because, together, they provide empirical supports for the checklist and the principles that are derived at the end of the thesis. The materials in each chapter are not limited to any one of, but can be a combination of, the following: literature review and its critique, conceptual analysis, empirical studies, statistical analyses, and discussion of findings. However, some experimental findings may lead to subsequent experiments. They may, therefore, be referenced or supported by the materials in a subsequent chapter. For the organisation of this thesis to be easily followed, a dissertation road map (Figure 1.1) and a summary of the six units of empirical studies conducted (Table 1.1) are thus provided. The diagram in Figure 1.1 gives an overview of the organisation of the materials in each chapter and their inter-relationships. The diagram has its own convention: a rectangular box represents content, analysis, review, empirical work, and/or method used. A rounded rectangular box represents an outcome or a product of studies or work carried out. There are two types of outcomes: research questions and findings. Research questions are used as a basis for the empirical studies in Chapters 3, 4, and 5. Findings from chapters 2 to 5 form a basis for the synthesis in Chapter 6. And finally, the arrows in Figure 1.1. denote relationships.

Chapter 2 first reviews the research in the Psychology of Programming and draws together the empirical findings fundamental to deriving a Model of Programming Process (MoPP) that is used to drive the first part of the research (Chapters 3 and 4). The model highlights two major areas to be tackled in the research: programming paradigm and perceptual coding. The literature in these two areas is further reviewed and a set of design principles for diagrammatic languages is summarised from this. Furthermore, the role of Visual Language on providing perceptual cues in visual programs is explored and a Visual Language Matrix (VLM) for visual programs is suggested. The chapter concludes with a set of research questions worth exploring, which are used as a basis for the empirical studies presented in Chapters 3 and 4.

Chapter 3 presents the experiment in Study unit 1 (see Table 1.1) that compared novices' performances between control flow programs and data flow programs. The study also provides evidence for the superiority of three visual programs over a convention textual program. Finally, it discusses and provides some evidence for an indication of paradigm preference among the students who had participated in the empirical studies carried out in this research.

Chapter 4 presents two studies relating to representation of flow and layouts in graphical programs. It comprises experimental studies in Study units 2 to 5 (see Table 1.1). The Maze study consisted of two experiments conducted to compare three directional representations: Arrow, Line, and Juxtaposition. The Flow study consisted of two experiments that compared a total of six visual program layouts, each requiring a different way to traverse a diagram.

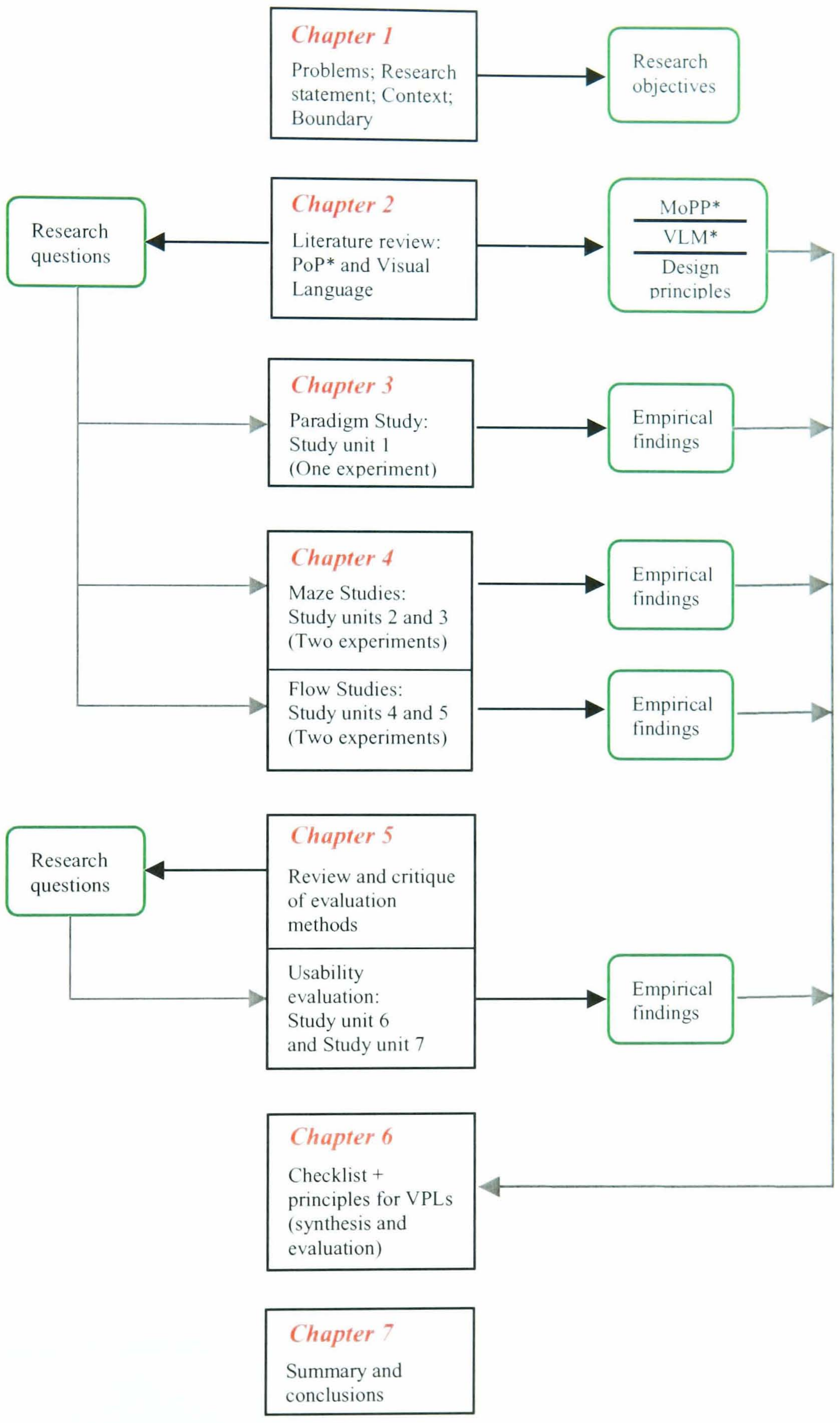
Chapter 5 reviews and critiques the literature on usability evaluation methods for their appropriateness to evaluating a VPL. It identifies a research question for which an evaluation of a commercial VPL, Prograph, was conducted using a qualitative inquiry approach. Findings are discussed, a further analysis of the empirical data is presented, and a framework for restucturing Cognitive Dimensions analysis is proposed. The applicability of the approach adopted in this chapter and the framework to other research contexts is also demonstrated.

Chapter 6 presents the process of deriving a checklist and principles for diagrammatic VPLs. It draws together and refines the results from the empirical studies presented in Chapters 3,4, and 5 and from the VLM of visual programs suggested from our analysis of the literature materials in Chapter 2.

Chapter 7 concludes this research. Contributions of the present research, its limitation and incompleteness, and avenues for future research are discussed.

Table 1.1 Units of empirical studies

Study unit	Name	Issues addressed	Research method/ Design	Data collection tools	Statistical methods
1	Paradigm study	Visual vs. Textual	Experiment/ Within-subjects	Visual Basic program; Questionnaire	ANOVA; t-tests; Descriptive statistics; McNemar-test
		Control flow vs. Data flow			
2	Maze study 1	Effect of Directional representation in non-programming context	Experiment/ Mixed-factorial	Visual Basic program; Questionnaire	ANOVA; Cohran Q-test; Descriptive statistics
3	Maze study 2		Experiment/ Mixed-factorial	Visual Basic program; Questionnaire	ANOVA; Cochran Q-test; Descriptive statistics
4	Flow study 1	Effect of traversal direction	Experiment/ Within-subjects	Visual Basic program; Questionnaire	ANOVA; t-tests; Cochran Q-test; McNemar-test; Descriptive statistics
5	Flow study 2	Effect of traversal direction and directional representation	Experiment/ Mixed-factorial	Visual Basic program; Questionnaire	ANOVA; t-tests; Power analysis, Discriminant Analysis; Descriptive statistics
		Cognitive ability vs. test performance	Choosing a Path Test (Ekstrom *et al.*, 1976)	Multiple-choice questions	Pearson correlation
6	Prograph study	Potential usability problem areas	Naturalistic inquiry; Immersion; Self-observation	Diary	Pareto analysis; Frequency statistics
7	HE-Plus study	Extending heuristic evaluation	Experiment/ Between-subjects	Usability problems report; Questionnaire	Mann-Whitney test; Kolmogorov-Smirnov test Descriptive statistics


Chapter 1
Problems; Research statement; Context; Boundary
Research objectives
Chapter 2
Literature review: PoP* and Visual Language
Research questions
MoPP*
VLM*
Design principles
Chapter 3
Paradigm Study: Study unit 1 (One experiment)
Empirical findings
Chapter 4
Maze Studies: Study units 2 and 3 (Two experiments)
Empirical findings
Flow Studies: Study units 4 and 5 (Two experiments)
Empirical findings
Chapter 5
Review and critique of evaluation methods
Research questions
Usability evaluation: Study unit 6 and Study unit 7
Empirical findings
Chapter 6
Checklist + principles for VPLs (synthesis and evaluation)
Chapter 7
Summary and conclusions

* PoP = Psychology of Programming; MoPP = Model of the Programming Process; VLM = Visual Language Matrix

Figure 1.1 Dissertation Road Map

2. PROGRAM, PROGRAMMING PARADIGM, AND PERCEPTUAL CODING

2.1 Introduction

A program is “a sequence of coded instructions which enables a computer to perform various tasks” (Collins New English Dictionary, 1998). Definitions of a program given in programming texts do not differ much, although they tend to include more technical terms such as, ‘instruction sets’, ‘algorithms’, ‘computation’, and so on. Generally, a program can be considered as a sequence of instructions for the computer to perform some calculations, to define some functions, objects or events, to describe the sequence of operations on objects, of events, and to describe flow of controls or flow of data. It seems clear that programming is a process of representing these instructions, descriptions, or definitions in the form that the computer can understand, with the representations used by programmers, and in the syntax of the programming language used. In short, a programming language provides programmers with a system of representation of various programming concepts.

This research focuses on usability issues in designing a VPL and explores the programming difficulties experienced by novices (see, for example Pane & Myers, 1996). There is ample evidence of novices’ difficulties with learning to program across various programming language constructs. Reviewing the literature on issues pertaining to psychological process of programming has enabled us to propose a model that represents this process, called ‘Model of the Programming Process’, or MoPP (Figure 2.1) helps us identify two major areas noteworthy to explore as a starting point. MoPP is described in the next section. The subsequent sections describe the information structure framework that is used as a foundation for MoPP, the areas of investigation relevant to our research as identified by MoPP, and other research relevant to exploiting perceptual coding and visual language for enhancing VPL usability.

2.2 Model of the Programming Process (MoPP)

This section describes MoPP , which is depicted in Figure 2.1 below.

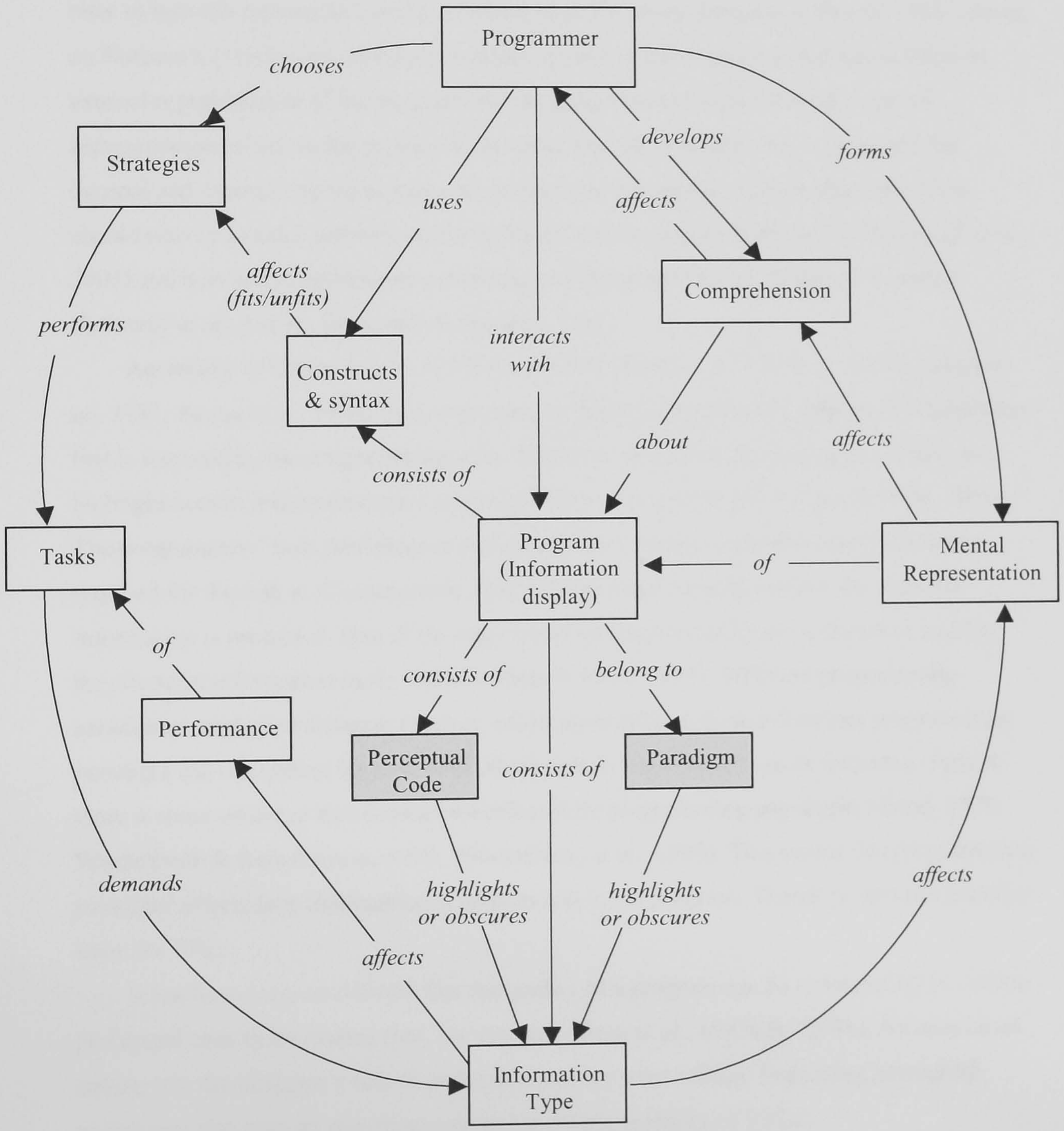


Figure 2.1 MoPP: Model of the Programming Process

A program is written by a programmer with the explicit aim of using a programming language (notation). Empirical evidence suggests that the programmer will do well if there is a cognitive fit between his/her mental representation and the external representation of the program (Green & Petre, 1996). External representations refer to symbols, notations or signs that stand for something or some aspect of the world (Eysenck & Keane, 1992). In this context, external representation refers to the program code. Internal or mental representations refer to how the represented world is perceived in the mind (Eysenck & Keane, 1992). Based on Norman's (1983) view of mental models, *mental representation* in this thesis refers to internal representation of the program that the programmer has and how that mental representation relates to the problem to be solved by the program. Not only should the internal and external representations of the program correspond to each other, but there should also be a match between representations and the programming tasks (Blackwell *et al.*, 2001) and between programming constructs and the programmers' preferred strategies (Soloway *et al.*, 1983a; Eisenstadt & Breuker, 1992).

According to Green & Petre (1996) and others (Sime *et al.*, 1977a & 1977b; Green *et al.*, 1981; Payne *et al.*, 1984; Gilmore & Green, 1984), a program is a display of information that is required by the programming tasks. Different programming languages or notations highlight certain information (in a program) while obscuring others (Green & Petre, 1996). The programmers' task performance depends on how readily accessible the information required for the task is. Consequently, their performance depends on how the required information is promoted. One of the tasks faced by notation designers is therefore making the obscured information more visible (Green & Petre, 1996). Different programming paradigms emphasise different types of information differently and therefore programming paradigm can also affect the ease with which certain information can be extracted. Indeed, there is some evidence that novices are affected by programming paradigms (Good, 1999; Wiedenbeck & Ramalingam, 1999; Wiedenbeck, *et al.*, 1999). This means that programming paradigm affects how information is emphasised in the program. Therefore, this is a usability issue for VPLs.

It has been long established that the quality of a program can be enhanced by providing perceptual cues to its readers (see, for example, Sime *et al.*, 1977a & 1977b). As mentioned before, it is the designer's task to make information more visible. Exploiting perceptual coding can also play an important role in improving usability of VPLs.

The following sections put forward a view of a program as an information display based upon the framework of information structure and presents research in the areas identified by MoPP. This view forms a basis for the work presented in Chapters 3 and 4. In this model, two major areas worth investigating for their relevance to this research are identified. These are the roles of programming paradigm and perceptual coding that affects how information is displayed in a program.

2.3 The information structure framework

Research into *program comprehension* attempts to explain how programmers understand programs, i.e. how they extract information from a program. There are at least three program comprehension models proposed so far: the top-down model by Brooks (1983), the bottom-up model by Pennington (1987), and the mixed model by Letovsky (1986). The programmers in the top-down model verify the hypotheses they made about the program based on the information in the program text. In Pennington's (1987) bottom-up model, the programmers' understanding of the whole program is built up from the information gathered from parts of the program text. They read the program text and extract different types of information from it. Their mental representation of the program is formed based upon the information extracted. The programmers in the mixed model comprehend the program opportunistically using both top-down and bottom-up approaches to extract the required information, depending on the cues available at the time. All three models consider a program as a display of information.

Findings by researchers into *programming knowledge* (syntactic/semantic knowledge, programming plans, and beacons) [see, for example, Soloway & Ehrlich, 1984] also support the notion of a program as an information display. Programmers employ different programming strategies in order to make the best use of their semantic and syntactic knowledge to construct the internal semantic structure of the program during program comprehension. Their programming knowledge is recalled from the long-term memory to be analysed in the working memory (Shneiderman & Mayer, 1979) and must be required by the program. This means that the program has to display the information required by the programmer.

According to some researchers, programmers use '*programming plans*' and '*beacons*' in helping their program comprehension and make it easy for programmers to recognise the functions of particular segments of code. Soloway & Ehrlich (1984) and Soloway *et al.*, (1983b) define '*programming plans*' as 'parts of a program code that represent certain stereotypical tasks'. Wiedenbeck (986) defines '*beacons*' as lines of codes that are used as typical indicators of a particular structure or operation. Indeed, the evidence of '*programming plans*' provided by Soloway & Ehrlich (1984) and of '*beacons*' provided by Wiedenbeck (1986) supports the notion of information display of programs. Experts do not study programs line-by-line. Their strategy is to look for '*programming plans*' and '*beacons*' in the program to verify their hypotheses about the program's functions. In other words, they seek specific information from the program very quickly with the aids of '*programming plans*' and '*beacons*'. Novices, however, study programs line-by-line and spend more time with programming syntax than high-level functions (Rist, 1986). They do not possess enough programming experience to be able to recognise programming plans as experts do. This means that '*programming plans*' are not represented well enough for the required information to be made accessible by novices without learning the strategies first.

Green & Petre (1996) summed up the findings of previous program comprehension research into two maxims of information representation, which form the backbone to understanding the psychological process of programming. The two maxims are described in the following sections.

2.3.1 The first maxim of information representation

> ***Every notation highlights some kinds of information at the expense of obscuring other kinds.*** *Not everything can be highlighted at once. If a language highlights data flow then it may well obscure the control flow; if a language highlights the conditions under which actions are to be taken, as in a rule-based language, then it probably obscures the sequential ordering of actions. Corollary: part of the notation design problem is to make the obscured information more visible.* (Green *et al.*, 1981)

In short, one notation may be better than another in representing certain information and therefore yields better performance on the tasks that require that information. The implication is that no one notation is best for all kinds of programming tasks. The first maxim is summarised from a number of empirical evidence for Match-Mismatch phenomenon (Gilmore & Green, 1984) and the dual model of mental representation of program (Pennington, 1987) as discussed below.

<u>The Match-Mismatch phenomenon</u>

Match-Mismatch is a phenomenon observed when the Match-Mismatch hypothesis is supported (Gilmore & Green, 1984). The Match-Mismatch hypothesis states that performance is best when there is a match between representation and information required by the task. Different tasks require different kinds of information. For example, to find out the sequence of some operations in a program, one needs control-flow information whereas to understand the changes in certain variable values, one needs data-flow information.

Answering what the effect of some conditions might be would require different information from answering what the conditions might be given the effect. Different notations (or representations) may emphasise different information. Therefore, an evidence of the Match-Mismatch phenomenon can be gained if it can be shown that the same notation yields different performance on different tasks or that different notations yield different performance for the same task.

Many Match-Mismatch phenomena have been observed with textual programs (Sime *et al.*, 1977a & 1977b; Green, 1977; Gilmore & Green, 1984; Sinha & Vessey, 1992) and, later, with visual programs (Green *et al.*, 1991; Good, 1999). The first evidence of the Match-Mismatch phenomenon came from the work of Sime *et al.* (1977b) whose programs were written in three procedural style micro-languages. Their studies focused on the design of conditionals and, therefore, the micro-languages were devised for their studies to suppress the language features other than conditionals such as assignment, iteration, and the use of logical operators and negation. The micro-languages were NEST-BE, NEST-INE, and JUMP styles (see Figure 2.2).

Sime *et al.* (1977b) compared response time performance of the three micro-languages on the same tasks: tracing the program backward and tracing the program forward. They found that the two NEST styles outperformed the JUMP style in '*programming*' (drafting a program) but that NEST-INE was the best in '*deprogramming*' (checking the program). Their explanation is that there are two types of information in conditional programs: *sequential* and *taxon* information. *Sequential* information gives the order of what the program does. *Taxon* information gives the conditions for certain actions. In a procedural language, '*programming*' requires translation of *taxon* information into *sequential* information. '*Deprogramming*' is the reverse process. There was no performance difference between the two NEST programs while both of them performed better than the JUMP program in '*programming*'. This, they explained, was due to indentation used in the two NEST styles, which provided redundant coding for sequential information. In '*deprogramming*', however, the NEST-INE outperformed the other two programs. The only explanation was that predicates that were *redundantly* repeated in NEST-INE style helped clarify *taxon* information. For example, in Figure 2.2, the NEST-INE notation used 'NOT green' instead of 'ELSE' as used in the NEST-BE notation. Their results lead to the Match-Mismatch hypothesis: that performance is best when representation (micro-language) matches the information required by the tasks.

JUMP

```
     IF hard GOTO L1
     IF tall GOTO L2
     IF juicy GOTO L3
     roast stop
L1   IF green GOTO L4
     peel grill stop
L2   chop fry stop
L3   boil stop
L4   peel roast stop
```

NEST-BE

```
IF hard THEN
BEGIN peel
   IF green THEN
   BEGIN roast
   END
   ELSE
   BEGIN grill
   END
END
ELSE
BEGIN
   IF tall THEN
   BEGIN chop fry
   END
   ELSE
   BEGIN
      IF juicy THEN
      BEGIN boil
      END
      ELSE
      BEGIN roast
      END
   END
END
```

NEST-INE

```
IF hard peel
  IF green roast
  NOT green grill
  END green
NOT hard
  IF tall chop fry
  NOT tall
    IF juicy boil
    NOT juicy roast
    END juicy
  END tall
END hard
```

Figure 2.2 Examples of micro-languages: JUMP; NEST-BE; and NEST-INE
(Sime *et al.*, 1977b, p. 112)

Gilmore & Green (1984) conducted an experiment comparing response time performance between procedural and declarative notations (micro-languages) and between programs with or without typographical cues such as indentation and white spaces. Participants answered forward and backward questions. Forward questions give the conditions and ask for the outcomes. Backward questions ask for the conditions of the given outcomes. Forward questions thus require *sequential* information while backward questions require *taxon* information, which they called *circumstantial* information. From here on, the term *circumstantial* will be used to refer to *taxon* information. Their results showed that:

1. In a procedural notation that they used in the experiment, programmers performed better when answering forward questions than backward questions. In other words, *sequential* information is easier to be extracted from a procedural notation than *circumstantial* information.

2. In a declarative notation that they used in the experiment, programmers performed better when answering backward questions than forward questions. In other words, *circumstantial* information is easier to be extracted from a declarative notation than *sequential* information.
3. No one notation was best in both types of tasks. That is, different notations highlight information differently.
4. Typographical cues were an effective means for accessing the information obscured by the structure of the notation.

The first two points give the evidence for the Match-Mismatch phenomenon in both procedural and declarative notations. The last point above supports our argument for the need to investigate the role of perceptual coding in enhancing program comprehension.

Nonetheless, the Match-Mismatch phenomenon was not always observed in graphical programs. In the study by Moher *et al.* (1993), the Nested Petri net program, designed to represent a procedural notation, exhibited much faster backward performance than forward performance. In fact, in all Petri net programs used in the experiment, backward tracing outperformed forward tracing. This implied that *circumstantial* information was easier to extract in a procedural notation. The Match-Mismatch hypothesis was therefore not supported for this specific visual program. Whitley (2000) speculated that this might have been due to poor design of the forward Petri net representation for the experiment by Moher *et al.* (1993). Interestingly, however, a similar result, challenging the Match-Mismatch hypothesis in visual programs, had also been reported by Curtis *et al.* (1989) and Good (1999).

In one of Good's (1999) experiments that investigated the match between tasks and representation for miniature control-flow and data-flow VPLs, the Match-Mismatch effect was found to be overridden by 'control flow supremacy'. In other words, the best performance was always achieved with control-flow tasks, regardless of representation, and with control-flow representations, regardless of tasks. In another experiment Good (1999), however, the Match-Mismatch effect was found only with accuracy data but not with response time.

The programs used in Curtis *et al.*'s (1989) study were similar to flow diagrams and were hence procedural. Table 2.1 gives the mean time taken per question for diagrams that used ideogram as graphical primitives for three spatial arrangements tested ['Sequential', 'Branching', and 'Hierarchical' (Curtis *et al.*, 1989)]. It shows that forward tracing is slightly faster than backward tracing for the 'Sequential' diagram only. It appears that there is no statistical difference between the two tasks in the programs used by Curtis *et al.* (1989).

Table 2.1 Response time performance for forward and backward questions

Ideogram +	Mean seconds per question (approx. reading)	
	Forward	Backward
Sequential	38	43
Branching	36	35
Hierarchical	39	36

(Estimated from plots in Curtis et al., 1989; Figures 4 and 5, p 183-184.)

Green *et al.* (1991) tested the Match-Mismatch hypothesis using the Boxes and the Gates notations of LabVIEW to represent *sequential* and *circumstantial* programs, respectively. Contrary to the previous results reported by Curtis *et al.* (1989) and by Moher *et al.* (1993), the Match-Mismatch phenomenon was observed in Green *et al.*'s (1991) experiment. Whitley (2000) commented that the studies by Green *et al.* (1991) and by Moher *et al.* (1993) differed in "the use of visual shapes (syntax) and in the semantics attributed to those shapes" and that the Petri net programs differed only in *secondary notation. Secondary notation* refers to redundant coding used as an extra means to improve the program beyond the 'official' semantics of the programming language (Green & Petre, 1996). In this case, the Petri net programs are different in layout and the arrangement of graphical primitives provides a means to convey information in addition to the primitives themselves, hence, *secondary notation*. The diagrams used by Curtis *et al.* (1989) also differed in *secondary notation* only because they differed in layout. Regardless of what could explain these conflicting results, it remains an open question whether the empirical findings based on textual programs are also applicable to visual programs.

The dual mental representation theory

In an empirical study, Pennington (1987) showed that programmers form two mental representations of program. The first representation developed by the programmers was text-based or a '*program model*'. The second mental representation was a '*domain model*'. A '*domain model*' refers to what the program text is all about and hence its functions. The programmers are said to have a '*program model*' or a '*domain model*' mental representation depending on their performance of the various information types implicit in the program. The dual mental representation theory supports the first maxim of information representation for two reasons: Firstly, the two mental representations developed by the programmers in Pennington's (1987) study were not developed simultaneously, but one after another. Secondly, the procedural information necessary for the first mental representation, i.e. the '*program model*', was highlighted by the procedural language used in her study (Pennington, 1987). Other information necessary for the second mental representation was obscured. At a later stage, through interactions with the program, functional information became better understood and the second mental representation was subsequently developed.

In procedural languages, programs are written in the sequence of execution so control-flow information is easier to extract than other kinds of information, such as functional information. In object oriented and data-flow languages, data are active and are passed to objects or functions to perform activities, which fire only when the required data are available. Therefore, in these languages control-flow information is obscured, and data-flow and function information is explicit. Based on Pennington's (1987) work, it is therefore expected that programmers' mental representation of programs are "*domain model*" for declarative, functional, data-flow, and object-oriented languages. In addition to Pennington's (1987) work, there has been other research conducted into the effect of programming languages on programmers' mental representation. These findings are given in Table 2.2. The column labelled 'Expected' refers to the expected mental representation. For example, in the first row, for a procedural language such as Pascal, procedural information should be more easily extracted from the program than functional information. Therefore, programmers' mental representation is expected to be a '*program model*'. On the other hand, non-procedural languages such as Prolog and C++ emphasising on functions and data and therefore a '*domain model*' mental representation is expected.

The data in Table 2.2 show that programmers' mental representation of program depends on at least two factors: the programming paradigm and programming experience. Novices develop the same mental representation of programs as expected. However, this differs with experts. Experts' mental representation of a program is not always what it is expected with the logic of the first maxim of information representation. It is a question of whether or not a programming paradigm truly affects novices' mental representation of programs. Therefore, another research question to answer is, what the role of a programming paradigm on program comprehension is? If novices are affected by programming paradigms, there is an implication in making design decisions for language designers. Questions that designers might ask themselves are: "Which paradigm to choose?", "What programming knowledge or information type is highlighted or obscured by the chosen paradigm?", "How to support or promote the information that is obscured by the paradigm?", and so on.

Table 2.2 Evidence of mental representation of programs

Program Experience	Language	Expected Mental Representation	Findings		Reference
			Program model	Domain model	
Novices	Pascal	Program model	✓		Corritore & Wiedenbeck (1999); Wiedenbeck *et al.* (1999)
	C	Program model	✓		Wiedenbeck & Ramalingam (1999)
	C^{++}	Domain model		✓	Wiedenbeck *et al.*(1999); Wiedenbeck & Ramalingam (1999); Davies (2000); Wiedenbeck & Ramalingam (1999)
	Control-flow VPL	Program model	✓		Good (1999)
	Data-flow VPL	Domain model		✓	Good (1999)
Experts	Fortran	Program model	✓		Pennington (1987)
	Cobol	Program model	✓		Pennington (1987)
	C	Program model	✓		Corritore & Wiedenbeck (1999); Wiedenbeck & Ramalingam (1999)
	Prolog	Domain model	✓		Bergantz & Hassell (1991)
	C^{++}	Domain model	✓	✓	Corritore & Wiedenbeck (1999); Wiedenbeck & Ramalingam (1999); Davies (2000)

2.3.2 The second maxim of information representation

When seeking information, there must be a cognitive fit between the mental representation and the external representation. *If your mental representation is in control flow form, you will find a data flow language hard to use; if you think iteratively, recursion will be hard.* (Green & Petre, 1996)

External representation of a program refers to the program code, i.e. programming syntax and language constructs and the overall look of the program. The second maxim thus suggests that we focus on how easy the programming syntax and constructs are for novices to use, i.e. how much extra efforts novices must make in order to write a program or to understand a program, which is the case when external representation does not match mental representation.

Both novices and experts benefit from a cognitive fit between the strategy imposed upon them by the programming language constructs and their preferred strategy. However, the difficulties incurred by the mismatch between programming language constructs and the preferred strategies are more severe among novices than experts. Novices found some programming language constructs difficult to use (Samurçay, 1990) because they were unable to implement the strategy that they would have preferred in real-life (Soloway *et al.*, 1983a and 1983b; Eisenstadt & Breuker, 1992). Furthermore, they have difficulties with determining which constructs to use and how to co-ordinate them 'as a unified whole' (Soloway *et al.*, 1983b).

Novices have difficulties with the assignment statement, initialisation, variables, logical operators, and negation. For example, in the statement *sum := sum + number*, a novice may wonder why the sum in the left-hand side of the statement should be the same as itself plus a number. The problem is that the two occurrences of the 'sum' variable in the statement refer to two different values (Samurçay, 1990). That is, the 'sum' on the left-hand side holds the current value while the 'sum' on the right-hand side holds the preceding value. Samurçay's (1990) empirical data also show that initialisation operation (e.g. *count := count + 1*) is more difficult than testing and update operations (e.g. *sum := sum + x*) because people do not usually have to carry out an initialisation process which involves using a variable, in manual execution of a problem. Variables impose another difficulty to novices. A variable represents an address in the register, which is an unfamiliar concept to novices. Novices found internal variables (variables used in programs) conceptually more difficult than external variables (input/output variables). This is because the values of internal variables depend on the internal states of the program while those of external variables can be controlled by the programmers (Samurçay, 1990). The logical operators AND and OR are also a frequent source of programming bugs. Novices used the OR operator less efficiently than the AND operator and when OR and Negation are used in a test expression, frequency of errors is high (Miller, 1974; Pane & Myers, 2000).

Iteration is another difficult programming concept for novices (Miller, 1974; Hoc, 1989; Samurçay, 1990). Samurçay (1990) defined iterative control structures in a program as being used to initiate "a response to problems whose solution requires the execution of identical actions/rules a certain number of times. The construction of an iterative plan involves the identification of the elementary actions/rules which must be repeated, and the condition governing end or continuation of the repetition". The major problem that novices have with iteration is that there is no cognitive fit between the way that novices prefer in performing iteration tasks and the strategy required by the programming language constructs. Soloway *et al.* (1983a) show that Pascal '*while*' loop imposes a different strategy from the strategy that novices prefer. Novices prefer the '*read/process*' strategy to the '*process/read*' strategy (see Figure 2.3) imposed by a typical Pascal '*repeat*' and '*while*' loops, respectively (Soloway *et al.*, 1983a; Samurçay, 1990).

When faced with an iterative coding task, novices construct a mental representation for execution sequence from their real-life experiences with iterative tasks (Eisenstadt & Breuker, 1992). However, this real-life mental representation cannot be easily fit into the programming language framework without restrictions. For example, in trying to employ their preferred strategy, the '*read/process*' (Figure 2.3), in their Pascal programs, novices create buggy programs due to the fact that the Pascal while loop facilitates the '*process/read*' strategy (Figure 2.3). Experiments by Eisenstadt and Breuker (1992) show that novices prefer to perform an iterative task in multiple passes over a set of data, i.e. doing one task at a time over the whole set of data. This suggests that they "think naturally in terms of temporal abstraction, and that the use of aggregate data objects is far simpler for them than the confusing detail required to specify temporal sequence. Hence, temporal abstraction may be the most natural way of expressing iteration". This hypothesis has, in fact, been supported by the work of Lewis & Olson (1987).

```
Example of 'read/process' strategy:
    Loop
      Do begin
        Read the ith value
        Test the ith value for exiting the loop
        Process the ith value
      End

Example of 'process/ read' strategy:
    Read the ith value
    While (Test the ith value)
      Do begin
        Process the ith value
        Read the (i + 1)th value
      End
```

Figure 2.3 '*Read/process*' and '*process/read*' strategies

Moreover, the '*while*' loop is more difficult to conceptualise than the '*repeat*' loop for novices. When asked to write a procedure in natural language most students in the experiment by Samurçay (1990) wrote loop-plans in which the order of operations was a description of '*actions/repeat mark/end control*'. Furthermore, when the exit condition is governed by the number of iterations known in advance, conceptualisation is easier than when the exit condition depends on a variable value calculated in the loop. Therefore, the '*for*' loop in BASIC may be easier than the '*while*' and '*repeat*' loops.

Recursion is another difficult concept for novices to master. It has been observed that successful learning of recursion depends on whether they possess an adequate mental model of recursion (Pirolli & Anderson, 1985; Kessler & Anderson, 1989; Kahney, 1992). There are some indications that novices who learn iteration first develop an adequate mental model for learning recursion and thus are more ready to learn recursion than those who learn recursion before iteration (Kessler & Anderson, 1989). Teaching novices iteration first might lessen the problem with recursion.

The research findings above show that novices' difficulties arise when there is no cognitive fit between the external and mental representation of programs or between real-world execution and the execution required by the programming language constructs. This, in effect, supports the second maxim of information representation. The implication to language design is that the programming language should provide language constructs that are natural to use as far as possible, i.e. the strategies required by the constructs should match novices' preferred strategies.

2.4 Programming paradigms

The second maxim of information representation calls for a cognitive fit between mental and external representations. According to Norman (1983), however, mental models are not stable as they can change or be forgotten. Consequently, the mental representation of a program could also change. When the newly developed mental representation is not the one preferred by the programmers, they may find the language hard to use. There is some empirical evidence that programming paradigms affect novices' mental representation of programs. The sections below address two relevant issues: a) the effect of programming paradigm on mental representation; b) the programming paradigm preference among novices.

2.4.1 Effects of programming paradigms on mental representation

Novices and experts seem to be affected differently by programming paradigm. According to Petre (1996), experts are not constrained by the underlying paradigm of a programming language when writing a program. They use strategies across paradigms in solving programming problems and then translate the solution into the target programming language. However, there seems to be some paradigm effects on experts in program comprehension. Results from single paradigm studies on expert programmers are inconsistent (see Table 2.2). The Prolog experts in Bergantz & Hassell's study (1991), and the C^{++} experts in Davies' (2000) and in Corritore & Wiedenbeck's (1999) studies exhibited a '*program model*' mental representation even though a '*domain model*' representation was expected. There is some within-study research that compared paradigm effects on comprehension. Wiedenbeck & Ramalingam (1999) compared comprehensibility of C and C^{++} by novice programmers. This study shows that the mental representation of the program of the more skilled novices does not change with the underlying programming paradigm while that of the less skilled novices does. Corritore & Wiedenbeck (1999) reported similar results for expert programmers performing comprehension and maintenance tasks of large C and C^{++} programs. Both C and C^{++} experts exhibited '*program model*' mental representation. They, nevertheless, stated that program size might have a stronger effect on comprehension than the paradigm, which was the reason offered as to why a '*program model*' was preferred with the C^{++} programmers and not a '*domain model*' as they had expected. Within-study empirical results on novices, on the other hand, have been consistent. Novices' mental representation of programs is program oriented for procedural languages [Pascal (Corritore & Wiedenbeck, 1999; Wiedenbeck *et al.*, 1999); C (Wiedenbeck & Ramalingam, 1999); and a control-flow VPL (Good, 1999)], and is domain oriented for C^{++} (Wiedenbeck *et al.*, 1999;

Wiedenbeck & Ramalingam, 1999; Davies, 2000) and for a data-flow VPL (Good, 1999). These results agree with the expectations concerning the first maxim of information representation discussed earlier.

So, the paradigm effect on comprehension for novices and experts can be summarised as follows:

1. Novices are affected by paradigm difference. Their performance depends on what is highlighted or obscured by the notation.
2. Experts are not always affected by paradigm difference but may be more strongly affected by program size.

2.4.2 Paradigm preference

From the evidence stated above, novices' mental representation of programs seems to be affected by programming paradigms. This is made more complicated if there exists a paradigm preference. When they work with the language in their preferred paradigm, novices mental representation would be affected positively and therefore, they would do better than otherwise.

There are some indications that novices may prefer the control-flow paradigm. In an experiment comparing the ability to write queries in SQL (nonprocedural query language) with TABLET (procedural query language), Welty & Stemple (1981) found that performance was better for difficult queries with the procedural query language than with the nonprocedural one. The C^{++} novice participants in Wiedenbeck *et al.*'s (1999) study (the less skilled group of novices) exhibited the same program model mental representation as the Pascal participants in the same study. Davies (2000) compared comprehension performance between experts and novices across all the five information types that were identified to exist in programs by Pennington's (1987) study. They are: *control flow*, *data flow*, *function*, *operation*, and *state* information. His data (Davies, 2000) for the novice group indicated that *control flow* performance was the strongest among all information types. Good (1999) compared novices' program comprehension performance between a control flow and a data-flow visual program written in a micro-language. Her results showed a '*control flow supremacy*' among novice participants. That is, overall novices' performance for the control flow VPL was higher than for the data flow VPL.

If the paradigm preference speculation is true, it has an implication on the design of programming languages for novices. For novices who find control flow languages easier than other types of languages, extra supports to aid them in the comprehension of non-control flow information will be required. Furthermore, according to the second maxim of information representation, when confronted with a non-control flow language, novices' performance may suffer because the language used does not have a cognitive fit with the type of language they prefer. This raises yet another research question: whether there is a paradigm preference among novices and, if so, which programming paradigm.

Choosing an appropriate paradigm for a VPL is not a straightforward matter. It would be ideal if programming language designers could choose one paradigm on merit of preference alone. If the hypothesis that novices have a control flow preference can be supported, a control flow language should be chosen over a data flow language. However, this is usually not the case as there is a paradigm shift from control flow to data-flow VPLs (Blackwell *et al.*, 2001). Depending on many factors, designers may not have a control over the paradigm choice but they could improve the usability of the programming languages by some other means. One way to do this is to exploit perceptual coding to enhance targeted or required information in the program so that cognitive demands on programmers can be lessened.

2.5 Perceptual Coding

Making programs "easier to write is to make them easier to read" (Green, 1980) because programming is an iterative loop of writing-reading-and-comprehending the program code. Programmers need to read the code to understand it in order to correct it. Perceptual factors are important for program understanding (Green, 1980). As Green (1980) put it: "When a train of thought is broken again and again by the need to find something out the hard way, it is difficult to piece the thoughts together into inspirations: it is difficult enough even to finish a simple train of thought without making a mistake, simply because of having to get the information in some tedious and error-prone way". Therefore, for the 'train of thought' to be finished smoothly, the programs should be easily readable. Enhancing the appearance of programs can improve their readability, legibility, comprehensibility, and maintainability (Marcus, 1992). From this point onward, the term *appearance* is used to refer to readability, legibility, comprehensibility, and maintainability. Readability concerns how easy it is for readers to read the words and how appealing they are while legibility concerns their visibility, i.e. how easy they are to be identified and discriminated (Bivins & Ryan, 1991; Marcus, 1992). Although these two terms are traditionally associated with text, they will be used here, with VPLs, as referring to how easy the graphical elements on the screen can be interpreted and how discriminable they are.

The role of typography as perceptual cueing in aiding text comprehension and in document design has been well established (see for example, Klare *et al.*, 1975; Payne *et al.*, 1984; Bivins & Ryan, 1991; Marcus, 1992; Baecker, *et al.* 1995). Typographical cues map the internal structure of the information display to its layout (Payne *et al.*, 1984) and thus enhance visibility of the internal structure of the textual information. Likewise, a program, as an information display, may also make use of typographical cueing to make its structure more visible. In fact, using indentation, white space, and colour is a common practice among professional programmers in documenting textual programs. Marcus (1992) outlined design principles for documenting computer programs. These principles covered various typographical issues ranging from font type, font size, word spacing, header, footer, use of symbols, to the use of a specific layout grid. However, these are not language design principles, nor are they for VPLs.

VPLs use any of these three types of visual representations: diagrammatic, iconic, and static pictorial (Burnett & Baker, 1994) and some text. Typographical cues are therefore not the only possible perceptual cues. It is desirable to know what cues are available to programming language designers for the improvement of the *appearance* of visual programs. To find out what the cues could be, issues relating to designing diagrammatic notations and the design of visual language (defined by Marcus (1992), as verbal and visual signs that convey meaning to the reader) are investigated. The former suggests desirable properties of the representations used in diagrams which, when coupled with the latter, helps identify a set of possible perceptual cues for visual programs and, hence, interesting research questions with respect to perceptual coding of visual programs can be subsequently raised.

2.5.1 How readers read diagrams

Winn (1993) described the process that readers read diagrams as a repetitive loop of forming the goals, locating the right diagram within the document, extracting the information, and evaluating whether the goals are reached. The basic scanning strategy to extract the information from diagrams is that readers decide where to look for the information relevant to their search goals. The success of this strategy depends on the readers' knowledge of the symbol convention of the diagram and of the content because it helps them decide what to look for next.

However, search involves two pre-attentive processes unaffected by individual's characteristics, domain knowledge, and the knowledge of the symbol systems. One process is discriminating one symbol from another. The other is configuring symbols into groups. These two processes affect the perceptual precedence of the symbols, thereby determining where readers look first. Based upon Treisman's *feature integration theory* (Triesman, 1988), Winn (1993) explains that when one symbol differs from others in only one feature (colour contrast, shape, size, orientation, location, etc.), the search is a parallel process and faster than a serial search that occurs when it differs from others by more than one feature. Hence, discriminability and configuration are important perceptual factors affecting search.

Spatial arrangement of symbols (i.e. how symbols are grouped and connected) affects how readers perceive symbol configurations and thereby search efficiency. Therefore it is important to investigate what perceptual cues can be provided to readers in order to enhance the role of discriminability and configuration and the effect of spatial arrangement on search efficiency.

2.5.2 Design principles for diagrammatic notations

Fitter & Green's Principles

Over two decades ago, Fitter & Green (1979) suggested five principles of how to make diagrams a good programming language by exploiting perceptual coding. Today, these principles still hold as will be discussed later. The five principles are:

1. Relevance

This principle states that the information to be represented in the diagram must be relevant to what is needed by its users.

2. Restriction

Restriction is the extent to which the notation can be reduced to a number of standard components so that they can be composed into a program in a structured way.

3. Revealing and Responsiveness

This principle refers to how well the notation reveals the inherent structure underlying the data and processes and how responsive the notation is to the manipulation of the data in such processes.

4. Redundant Recoding

This principle refers to providing extra (redundant) means to represent the information so that performance can be improved.

5. Revisability

The final principle, Revisability, refers to how easy the diagram can be changed upon modification.

Cognitive Dimensions of Notations

To our knowledge, Fitter & Green's (1979) principles stated above have not been explicitly or directly applied to any research since. Nevertheless, we observe that these principles form a root to some of the dimensions in the Cognitive Dimensions of Notation (CDs) proposed by Green (1989) to be used to evaluate usability of information artefacts. In fact, the CDs framework has been used to evaluate programming languages by various researchers (e.g., Modugno, 1996; Green & Petre, 1996; Clarke, 2001; Cox, 2000). The framework consists of fourteen dimensions (or criteria) that provide evaluators with a discussion tool that helps identifying potential usability problems experienced by users of the programming language being evaluated.

The dimensions are: Abstraction gradient; Closeness of mapping; Consistency; Diffuseness; Error-proneness; Hard mental operations; Hidden dependencies; Premature commitment; Progressive evaluation; Provisionality; Role expressiveness; Secondary notation; Viscosity; and Visibility. We describe each dimension by giving selected example questions relevant to programming languages in Table 2.3. We quote these questions directly from a paper on VPL usability evaluation by Green & Petre (1996) and from the questions in the CDs Questionnaire designed by Blackwell & Green (2000) because we feel that they describe the dimensions more effectively and efficiently than definitions of the vocabularies in prose. These descriptions are later used in the Prograph study described in Chapter 5 during content analysis of the empirical data.

Table 2.3 Description of the dimensions in CDs in programming context
(Green & Petre, 1996 and Blackwell & Green, 2000)

Dimensions	Selected example questions for each dimension
1 Abstraction gradient	• Does the system give you any way of defining new facilities or terms within the notation, so that you can extend to describe new things or to express your ideas more clearly or succinctly? • What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
2 Closeness of mapping	• What "programming games" need to be learned? • Which parts seem to be a particularly strange way of doing or describing something?
3 Consistency	• When some of the language has been learnt, how much of the rest can be inferred? • Are there places where some things ought to be similar, but the notation makes them different?
4 Diffuseness	• How many symbols or graphic entities are required to express a meaning? • What sorts of things take more space to describe?
5 Error-proneness	• Does the design of the notation induce "careless mistakes"? • Do you often find yourself making small slips that irritate you or make you feel stupid?
6 Hard mental operations	• Do some things seem especially complex or difficult or difficult to work out in your head? • Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?
7 Hidden dependencies	• If … some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kinds of dependencies are hidden? • Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
8 Premature commitment	• Do programmers have to make decisions before they have the information they need? • Can you *(the programmers)* go about the job in the order you like, or does the system force you to think ahead and make certain decisions first?
9 Progressive evaluation	• Can a partially complete program be executed to obtain feedback on "How am I doing?"
10 Provisionality	• Is it possible to sketch things out when you are playing around with ideas, or when you are not sure which way to proceed?
11 Role expressiveness	• Can the reader see how each component of a program relates to the whole? • Are there some parts that you really don't know what they mean? What are they?
12 Secondary notation	• Can programmers use layout, colour, and other cues to convey extra meaning, above and beyond the "official" semantics of the language?
13 Viscosity	• When you need to make changes to previous work, how easy is it to make the change? Why? • Are there particular changes that are more difficult or especially difficult to make? Which one?
14 Visibility	• Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to know in what order to read it? • What kinds of things are more difficult to see or find?

Not all the dimensions correspond to the five principles by Fitter & Green (1979) or are specific to diagrams. Table 2.4 shows the correspondence between the principles and the dimensions in CDs.

Table 2.4 Direct correspondence between Fitter & Green's (1979) principles and the dimensions in CDs

Fitter & Green's (1979) principles	Corresponding dimensions in CDs (Green, 1989)
Revisability	Viscosity
Restriction	Abstraction Gradient
Redundant Recoding	Secondary Notation

Revealing and Responsiveness	Role Expressiveness;
	Visibility;
	Progressive Evaluation

Recently, Britton & Jones (1999) used CDs to identify six common properties for 'ease of understanding' of diagrams used in software specification languages. The six properties are number of symbols, consistency of symbols, discriminability of symbols, the degree of motivation of symbols, and amount of structure in the language, and the extent to which human perception is exploited. They recommended the following:

1. Use appropriate number of symbols in a diagram.
2. Different symbols should conform to a pattern of form or meaning.

   Example of consistent symbols are = and ≠.
3. Different symbols should be easily distinguishable from each other.

   It is recommended that discriminability level be raised by the use of different sizes, fonts, shapes, shading, and colour.
4. Use appropriate level of clear and visible abstraction.
5. Match symbols to real world objects or concept.
6. Exploit human visual perception with the help of perceptual cues and *secondary notation.*

The principles from the work of Britton & Jones (1999) and of Fitter & Green (1979) and the dimensions in CDs by Green (1989) share some common grounds. We therefore amalgamate them to come up with a set of design principles for diagrammatic programming languages as follows:

Principle 1: Provide appropriate means and level of abstraction.

In order to gain optimum number of symbols (as suggested by Britton & Jones (1999)) and 'Diffuseness' in CDs), the notation or the programming language should allow some abstraction by having a number of standard components that can be composed into a program in a structured way ('Restriction' in Fitter & Green (1979)). The abstraction or restriction level should be optimum so that they are visible and easy to understand ('Abstraction Gradients' in CDs and 'Amount of structure' in Britton & Jones (1999)).

Principle 2: Use clearly distinguishable, familiar, and revealing representations and names.

Symbols or graphical elements should be 'Revealing and Responsive' (Fitter & Green, 1979). That is, they should be visible ('Visibility' in CDs), easily discriminated ('Discriminability of symbols' in Winn (1993) and in Britton & Jones (1999), not error-prone ('Error-proneness' in CDs), and role expressive ('Role expressiveness' in CDs). In order to be revealing, Britton & Jones (1999) suggested that the symbols used should conform to a pattern of form and meaning ('Consistency of symbols' in Britton & Jones (1999) and 'Consistency' in CDs) and match the represented objects or concepts in the real world ('Degree of motivation of symbols' in Britton & Jones (1999) and 'Closeness of Mapping' in CDs) and that 'Human visual perception' be exploited by using appropriate perceptual cues. We suggest that these recommendations apply to words that are used in naming programming objects as well.

Principle 3: Use secondary notation as appropriate.

Providing more than one means to convey information can help improve the ease of understanding of the notation ('Redundant recoding' in Fitter & Green (1979); 'Secondary notation' in CDs; and 'Exploit human perception by using secondary notation' as suggested by Britton & Jones (1999)). However, secondary notation should be used with care because there is evidence that it does not always help (Petre, 1995). For example, using many different colour schemes as a second means to provide information in the program in addition to the official program code might increase cognitive demand on readers.

Principle 4: Support modification through simplicity, clarity, and flexibility.

Changes to the program should not be difficult ('Revisability' in Fitter & Green (1979); and 'Viscosity' in CDs). This implies that simplicity and clarity of the design elements should be enforced and that dependency between entities in the notation should be made explicit (the 'Hidden dependency' dimension in the CDs). Where dependency is inevitable, it should be made visible ('Visibility' in CDs). Modification can also be supported if the programming language provides some means for low fidelity activities, such as sketching, to "play around with ideas" (Blackwell & Green, 2000) — i.e. the 'Provisionality' dimension in CDs. This implies flexibility.

Principle 5: Support evaluation

A good programming language should support opportunistic design — as Green (1990) put it, "Today's view is that program design is exploratory, and that designs are created opportunistically and incrementally". It is thus an iterative process of modification and evaluation until the programmer gets it right. To support this is to provide some functionality that allows small sections of code to be tested or animated for the evaluation of the unfinished programs ('Progressive evaluation' in CDs).

Principle 6: Offload cognitive efforts required where possible

This is to avoid programming concepts or representations that are difficult to understand or handled within the capacity of the short-term memory ('Hard mental operations' in CDs) or that requires that the programmer to look ahead ('Premature commitment' in CDs)—that is to anticipate what would happen if certain code is implied before it is actually written.

We do not include the principle 'Relevance' suggested by Fitter & Green (1979) in our list here because it is not an absolute necessity. It is common practice among programmers that information represented is relevant, i.e. source code, comments, and information about the program and programmers. Furthermore, occasionally putting all relevant information on one screen can adversely affect the *appearance* of the program. For example, comments are relevant information in a program but to what extent should they be presented?

The first three principles highlight the need for perceptual cueing in diagrammatic programming languages. Abstraction is difficult to represent and optimum amount of abstraction (Principle 1) is also hard to be achieved. Too much abstraction makes it hard for novice users to understand the code. Too little abstraction raises the number of representations or symbols used in the language to be learned beyond the 'magic number seven plus or minus two' (Miller, 1956) and hence the mental load imposed upon users. Good use of perceptual cues can help the symbols represent abstractions more efficiently. By paying attention to discriminability, to consistency of use, to visibility of dependency, and, where possible, to using familiar representations, the representations can be made more revealing (Principle 2) and program *appearance* be improved. Providing redundant (extra) perceptual cues can also help enhancing program *appearance* when appropriate (Principle 3).

2.5.3 Visual language

The preceding sections investigated issues surrounding diagrammatic notations suggested that the *appearance* of visual programs could be improved by the exploitation of perceptual cues. This section explores visual language design of documents and applies it to visual programs in order to derive a set of perceptual cues that can be used to enhance the *appearance* of visual programs.

Program is an information display and can be considered a kind of document. Visual programs can, similarly, be considered a special kind of document, of which information is represented mainly by graphical elements. In good text-based documents, readability and legibility are of utmost important for the information to be easily understood by readers. Likewise, a good visual program should be easy to read and to be interpreted and the graphical elements should be appealing, easily identified, and discriminated. Because careful design of visual language in text-based documents can improve readability and legibility (Marcus, 1992), analysing visual language design for text-based documents, therefore, might help us understand how visual language design of visual programs could improve their readability and legibility, and hence, their *appearance*.

Visual language refers to 'all the verbal and visual signs that convey meaning to a viewer' of the documents (Marcus, 1992). Typically, these signs fall into the following categories: typography, colour, layout, and symbols. The combination and overall effect of these signs makes up the visual language of a document. Each document thus has different visual language. Marcus (1992) provides numerous guidelines for designing a good graphical user interface (GUI) in each of the following categories: layout, typography, colour, symbolism, charts and diagrams, and screen design. However, due to the sheer amount that is available, these guidelines are too general and hence, not always easy to use. For example, it is not easy to know which guidelines one should use and in which situation they should be used. Guidelines that work in one situation may not work in another. For example, the guideline to avoid excessive use of colour may apply very well at a page or screen level, but may not be applicable at document or web site level where background colour may be used as section divider. Similarly, there are plenty of user interface design principles for designers to choose from. However, design principles may conflict one another and tradeoffs have to be considered. Therefore, both guidelines and principles are too general and difficult to apply. Designers must rely on their experience to a large extent. What is needed is a structured and holistic view of the document concerned in order to apply guidelines and principles in context and coherently across different levels of the document.

By taking a structured and holistic approach, the possibility that something is omitted could be reduced. We begin by looking at the design of professional (textual) documents — the whole document, not just one page, in order to ensure as complete a coverage as possible in our investigation. The following section describes how visual language of textual documents can be structured, what design elements and principles are appropriate within the structure. This knowledge will then be applied to visual programs in order to derive a list of potential perceptual cues relevant to the local and to the global design of visual programs in subsequent sections.

Text-based documents

Kostelnick & Roberts (1998) take a structured approach to visual design for paper documents to provide a basis for other forms of communication tools such as web sites and business presentations, using any medium other than paper. According to them, each document has its own visual vocabulary that makes up the visual language of that document which differs from that of other documents. Visual vocabulary includes textual and visual property ranging from typeface, size, shape, texture, to pictures on the page or screen. They (Kostelnick & Roberts, 1998) propose a framework called, Visual Language Matrix (VLM), to be used as a tool to systematically describe visual vocabulary of professional documents and to analyse how well the vocabulary helps the document serve its purpose, readers, and context of use. The matrix consists of four levels and three coding modes. The four levels distinguish between levels of design – from small to large-scale design decisions – depending on the level of granularity of design focus. The four levels are Intra, Inter, Extra, and Supra and are briefly described (Kostelnick & Roberts, 1998) as follows:

- Intra-level refers to "local variations of text, character by character, word by word".
- Inter-level refers to thing that "helps readers comprehend the text – line to line, paragraph to paragraph, column to column".
- Extra-level "includes pictures, data displays ... icons, and symbols ... may include some text to help readers understand them".
- Supra-level "includes top-down design elements that visually define, structure, and unify the entire document, whether print or electronic".

For each level, there are three modes to be considered. These are: textual, spatial, and graphic modes. Each cell in the matrix consists of design elements or visual vocabulary that make up the visual language of the documents as can be seen in Table 2.5. The four levels in the table are colour-coded in blue, green, red, and purple for Intra, Inter, Extra, and Supra levels, respectively. The coding scheme is used for ease of referencing because some of the design elements in this table will appear in a subsequent table.

The above framework and the design elements given in Table 2.5 are only applicable to textual documents. Even though they cannot be applied to visual documents such as visual programs directly, the framework (VLM for documents) and the design elements in Table 2.5 are used as a working template for deriving a VLM for visual programs in the next section.

Table 2.5 Visual Language Matrix (VLM) for documents
(Kostelnick & Roberts, 1998)

Level	Textual mode	Spatial mode	Graphic mode
Intra	▪ font type ▪ font size ▪ case (upper/lower) ▪ treatment (italic/ bold)	▪ spacing between characters ▪ spacing between words ▪ vertical spacing (superscript, subscript)	▪ punctuation marks ▪ symbols ($, £) ▪ treatment (underline, strike through)
Inter	▪ headings, levels of headings ▪ number or letters that signals in lists	▪ paragraphs, indentation, hanging indents, lists ▪ justified vs. unjustified centered text ▪ line lengths, margins, text arranged in tables, organizational charts, decision trees ▪ leading	▪ bullets and other listing devices ▪ gray scales highlight text ▪ linework in tables, organization charts, decision trees
Extra	▪ labels, call-outs, and captions for pictures and data displays ▪ numerical labels on *x*-and *y*-axes of data displays ▪ legends for data displays	▪ data displays: size of plot frame (*x*-and *y*-axes), orientation of plot frame (vertical or horizontal); space between bars, lines ▪ pictures size, viewing angle, perspective	▪ line weights or shading on pictures or data displays (bars or lines on graphs, gridlines, tick marks) ▪ details on pictures—line drawing vs. photograph ▪ use of color for pictures or data displays
Supra	▪ page headers or footers ▪ navigational bars ▪ major section or chapter heading or numbers ▪ tab labels—internal and external to the page ▪ titles on the cover or the spine of the document ▪ initial letters signalling the start of an article or major text segment	▪ shape, thickness, and size of the page (8 ½ x 11, legal size, scrollable length of the screen) ▪ orientation of the field (portrait vs. landscape) ▪ section dividers ▪ embossing ▪ placement of data displays and pictures in the document	▪ color or texture of paper ▪ page borders ▪ boxes, lines, or gray scales around pictures or data displays ▪ pictures or icons placed behind the text or spread over the whole document for cohesion ▪ lines in page headers or footers

Visual programs

We derive a Visual Language Matrix (VLM) for visual programs based upon the VLM for documents in Table 2.5. In VLM for documents there are four design levels. The Intra, Inter, and Supra levels are characterised by level of granularity of design focus. The Extra level concerns the design of graphical or visual entities such as charts, icons, symbols, etc. Visual programs are a specific kind of document, which consists mainly of graphical elements. Text is used sparingly in, for example, naming, listing, and commenting. Therefore, it is not necessary to have the Extra level as in VLM for text-based documents. VLM for visual programs thus consists of three levels (Intra, Inter, and Supra). The following briefly describes the three design levels.

- Intra-level design concerns design consideration at the most local level, i.e. local variations of graphical elements.
- Inter-level concerns the design of graphical elements and their relationships within one screen.
- Supra-level design refers to large-scale design of the whole program

The three modes: textual, spatial, and graphic are still applicable to the VLM for visual programs. Graphical representations used in visual programs require some text (textual mode) such as in naming of operations, can be arranged in many ways (spatial mode) such as in flowcharts, and can have variations in their design (graphic mode) such as in symbol shape and line thickness.

We obtain a VLM for visual programs (see Table 2.6) by going through the design elements (e.g., font size) in each cell of the VLM for documents (Table 2.5) one by one and consider their applicability to visual programs. The VLM for visual programs is a matrix of nine cells (3 levels and 3 modes). We transfer the design elements in the original VLM (for documents) to their corresponding cells in the new VLM (for visual programs) as appropriate. For each cell of the new VLM, we also add relevant design elements that do not exist in the original VLM but that could help make information more obvious. The colour coding scheme in the original VLM (Table 2.5) applies to the design elements in the new VLM (Table 2.6), where the additional black colour represents the elements not existing in the original VLM.

Some design elements in the Extra level of the original VLM (colour coded in red) appear in the Intra level of the new VLM because they refer to graphical visual objects which correspond to what the Intra level of the new VLM refers to, i.e. local variations of graphical elements.

Table 2.6 Visual Language Matrix (VLM) for visual programs

Level	Textual mode	Spatial mode	Graphic mode
Intra	▪ font properties (font type; font size; case; treatment) ▪ names/labels ▪ comments, error messages, and dialogs (call-outs)	▪ picture/icon size ▪ viewing angle ▪ orientation of plot frame (horizontal/vertical)- as in LabVIEW control panel ▪ perspective ▪ size of plot frame (x-and y-axes)	▪ punctuation marks ▪ symbols ($, £) ▪ treatment ▪ shading ▪ details of pictures/icons ▪ use of colour (colour coding) ▪ shape of icons/objects ▪ tool tip
Inter	▪ number or letters that signals order or sequence e.g. in trees to indicate traversing path	▪ scrollable length of the window/view ▪ layout (visibility aspect – leading, space between line (entries) and objects ▪ layout (structural aspect - traversal direction in reading diagrams)	▪ highlight ▪ linework in tables, organization charts, decision trees ▪ lineweight (broken, solid) ▪ shading ▪ use of colour ▪ bullets and other listing device ▪ scroll bar ▪ framing device (frames, boxes, lines)
Supra	▪ text in navigational bars ▪ numbers/letters that signal branches of control constructs e.g., yes/no arm, case ▪ text in call-graphs and data structure trees	▪ shape and orientation of windows/views unique to particular functions ▪ (consistent) position of objects across windows/ views	▪ background colour or texture of pictures/ icons ▪ boxes and lines around pictures or objects for reference to other parts of the program ▪ pictures or icons spread over the whole document for cohesion (i.e., icons, symbols on top bar of sub-window for reference to other parts of the program ▪ animation in training and debugging ▪ linework in call-graphs and data structure trees

(Items taken from VLM for documents in Table 2.5 are in blue-from Intra; in green-from Inter; in red-from Extra; and in purple- from Supra. Letters in black are new items added or descriptive comments.)

Only a few design elements from each of the other three levels in the original VLM are applicable to the new VLM. These are, for example, font properties, punctuation marks, and symbols which remain in the Intra level in both VLMs. Spacing between characters and words are not included in the new VLM because their contributions become much less significant in visual programs than in textual programs because coding in visual programs minimises text usage to, e.g. naming, and commenting. Likewise, design elements in the Inter level of the original VLM such as levels of headings, paragraphs, indentation, margin, etc. have no significance in visual programs while number and letters signalling in lists in textual documents can be used to signal the sequence or traversing path in a visual program in both Inter (sequence within a window or view) and Supra levels (sequence across windows or views). Most of the design elements in the Supra level of the original VLM relevant to visual programs are transferred across to the same level in the new VLM except for the scrollable length of the screen which we feel more appropriate for the Inter level as the programming elements are still seen within the same window or view.

In addition to the above, other design elements in black colour have been added into the new VLM such as icon shape — e.g., the diamond shape in flowchart representing decision point. Layout is an important design element that has been added to the Spatial mode of Inter level because layout affects visibility of programs and is affected by the programming language. Visibility is affected by proximity and links between objects, which could lead to spaghetti or jungle-gym programs. Programming languages or notational systems govern how the structure of the program (e.g., nested-if structure) is represented. Such representation in turn determines the traversal path that programmers must take when tracing the program. Therefore, the role of layout in visual programs on legibility and comprehension is not insignificant.

Perceptual cues for visual programs

From the VLM in Table 2.6, we obtain a list of perceptual cues that can be used to enhance the visual language in visual programs in Table 2.7. Across all three design levels, perceptual cues in textual mode can be obtained mainly from the variation in font properties used in names and labels and in signalling sequences. Typography does not play a significant role in visual programs as much as in textual programs.

In graphical mode, there are many cues ranging from using familiar objects, framing devices, highlights, animation, to variation in shape, thickness, shading, and colour of graphical representations. These are cues supporting the second design principle (*use clearly distinguishable, familiar, and revealing representation and names*) for diagrammatic languages that we summed up in Section 2.5.2. There are plenty design recommendations in this regard (see for example, Marcus (1992)). Graphic designers are often recommended to vary these properties in order to improve Figure-Ground contrast and Grouping (Kostelnick & Roberts, 1998). Good Figure-Ground contrast enhances discriminability of visual objects while Grouping of visual objects helps convey information on the relationships between them and can be done with framing devices. Examples for visual programs are representations of functions and loops in which their algorithms are encapsulated within a framed box that can be blown up to a larger size.

In spatial mode, certain drawing properties such as viewing angles could be used as perceptual cues, particularly where three-dimensional representations are used. For typical diagrammatic languages where representations are merely two-dimensional, however, the roles of layout and scrollable length are more significant. Scrolling can affect visibility of the information required and therefore increase mental load during searching. The less the users have to scroll for information, the better it is. Layout affects the program *appearance* (previously defined as referring to readability, legibility, comprehensibility, and maintainability) due to variation in visibility of the graphical objects and the way program structures are represented. In diagrammatic languages, layout is governed by the placement of graphical representations to express relationships between programming entities and representations of data flow and control flow. Depending on how these entities are placed, visibility and hence legibility of the program can be affected. Spatial arrangement of programming entities affects search in diagrams because it affects how readers of the visual programs perceive symbol configurations (Winn, 1993). Therefore, representation of flow which governs the order that programming entities must be traversed during searching is worth investigated.

Table 2.7 Design elements providing perceptual cues for visual programs

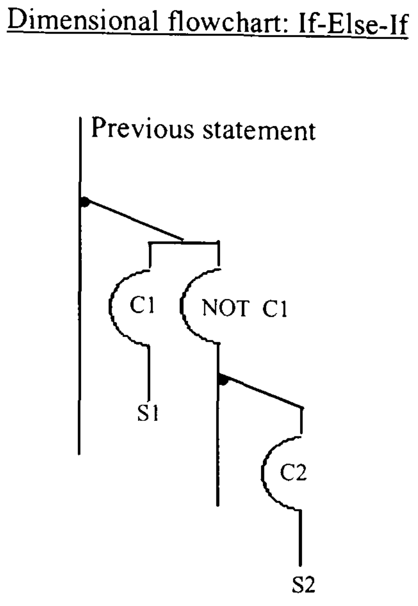
Mode	Perceptual cues
Textual	1. Font properties (type, size, case, treatment) 2. Names/labels for programming objects, navigational bars, and nodes 3. Numbers or letters signalling sequence or order of branches/cases
Spatial	1. Drawing properties (size, viewing angle, orientation, perspective) 2. Layout - Visibility aspect (arrangement of objects to get clear visible layout) - Structural aspect (representation of traversing path – line, arrow, nearness, adjacency) 3. Window/view properties (Scrollable length, shape, orientation, consistent position of objects across windows and views.
Graphics	1. Familiar objects and detail within objects where appropriate: - Symbols - Icons - Pictures - Listing device, e.g. bullets 2. Windows objects and tools such as hour glass, tool tip, navigational bars and scroll bars 3. Shape of graphical objects - Representing abstraction - shape variation is for coding (shapes have meaning –use standards e.g. the diamond shape for decision in flowcharts; where there is no standard, designer's choice) - Representing concrete objects-matching shapes to the represented real world objects 4. Framing device (lines/frames/boxes/windows) for discriminability and aesthetic reasons (e.g., encapsulation to draw attention or to group function code, e.g. LabVIEW structure nodes, Prograph use of windows 5. Thickness of line/solid/broken/patterned lines/frames - For coding - For discriminability and aesthetic reasons 6. Shading - For coding, e.g. to represent relative quantity for comparisons - For discriminability and aesthetic reasons 7. Use of colour (both background and foreground) - For coding, e.g. LabVIEW uses colour coding for data type (sometimes redundant recoding) - For discriminability and aesthetic reasons, e.g. to call for attention 8. Highlight/reverse video for emphasis 9. Animation for training and debugging

2.5.4 Representation of program flow

To understand a program represented diagrammatically, readers must traverse the diagram. Diagrams representing programs use different means of perceptual coding to represent the flow of data and of control. For example, flowcharts use connectedness and Nassi-Shneiderman diagrams (Nassi & Shneiderman, 1973) use insideness (Fitter & Green, 1979). Some diagrams have an inherent directionality, i.e. the direction of the easiest traversal. We shall call the direction in which a representation is most easily traversed from start(s) to ending(s), '*traversal direction*'.

'*Traversal direction*' varies among different notations. Some notations, such as spreadsheets and decision tables, do not have any particular direction of the easiest traversal. However, in some cases, where dependencies between different cells exist, they do have the easiest traversal direction. For example, in a spreadsheet, a cell may contain a formula referencing the value in another cell, which in turn referencing the value in a third cell. Calculating the value of the first cell from the formula is easier than finding out which formula in the spreadsheet uses the value of the third cell. In this case, traversing in the direction from the referencing cell to the referenced is the easiest.

Conventional flowcharts and the diagrams used in some VPLs, such as LabVIEW and Prograph, are traversed by following the links between nodes. Structured flowcharts, on the other hand, follow rigid rules for composing and traversing graphical objects. In a Bowles or a Jackson diagram (Bowles, 1977; Jackson, 1975) a left node and its sub-trees are traversed before a right node and its sub-trees. In a Dimensional flowchart and a Rothon diagram (Witty, 1977; Rothon, 1979) an upper node and its branched off descendants are traversed before a lower node and its branched off descendants. Jackson, Bowles, Witty, and Rothon diagrams all have a so called '*fall back*' feature (Green, 1982) which can occur during the tracing of a diagram. When tracing a diagram forward up to an operation at the end of a branch, one must '*fall back*' to the previous node and continue tracing a descendant node. If there is no further node, then one is supposed to '*fall back*' again (see examples in Figure 2.4). In these notations, the '*Restriction*' (Fitter & Green, 1979) level is high, and the programs are thus more tractable. However, whether or not they are easier or harder to use is difficult to answer without some empirical evidence. Green (1982), who conducted a detailed analysis on this issue, speculated that '*fall back*' would be difficult for novices.



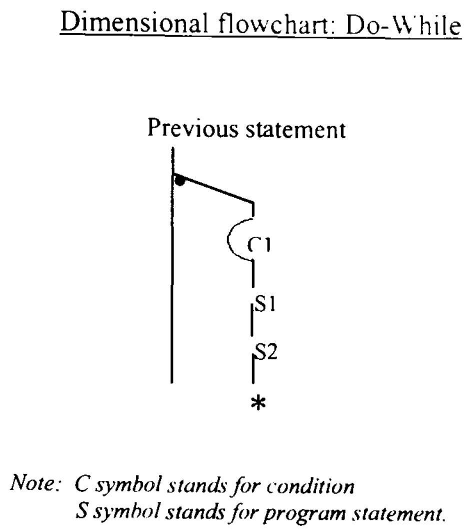

Dimensional flowchart: Do-While
Previous statement
C1
S1
S2
*
Note: C symbol stands for condition
S symbol stands for program statement.

Figure 2.4 Examples showing '*Fall Back*'

Traversal direction affects program comprehension in Petri net programs and flow chart style programs (Moher *et al.*, 1993; Curtis *et al.*, 1989). A study by Curtis *et al.* (1989) on the performance of expert programmers on nine different combinations of symbology and spatial arrangements ('Sequential', 'Branching', and 'Hierarchical') found that the 'Branching' arrangement was better than the 'Hierarchical' arrangement in tracing the program forward (giving conditions and asking for the outcomes), but not the other way around. Green (1982) reasoned that this might be due to the '*fall back*' feature of the 'Hierarchical' arrangement, which imposed cognitive demand on the readers and hence made forward tracing more difficult than 'Branching' arrangement. Hence, studying the effect of traversal direction has implications on the design of representation of program flow.

In diagrammatic VPLs, program flow is represented by symbols (e.g., shapes and lines) and the traversal direction. Historically, graphical symbols for connectedness as representation of program flow are arrow or line. However, some other systems such as Boxchart (Jonsson, 2001), BridgeTalk (Bonar & Liffick, 1990), or the Blox methodology proposed by Glinert (1990) juxtapose boxes or icons together. Thus we will investigate the following:

- The effect of directional representation on tracing a visual program.
- The effect of traversal direction on tracing a visual program.

2.6 Chapter summary

Our review on research in Psychology of Programming (e.g., program comprehension, programming knowledge, mental representation of program, etc.) has led to the Model of Programming Process or MoPP (Figure 2.1), which sums up the findings in this area. It shows the relationships between various entities in the programming process. The model is based on the information representation framework of programs that looks at programs as information displays. In the model, the entity *program* is the main focus. The *programmer* interacts with the *program* by employing some programming *strategies* in order to perform some programming *tasks*. However, The program is written in a programming language which consists of some programming *constructs* and which has its own syntax. These *constructs* and the programming language syntax are used by the programmer when writing the program to accomplish certain tasks. However, the constructs and syntax made available to the programmer by the language can affect the strategies that are actually used for the tasks in different ways. To enhance the ease of coding, they should have a cognitive fit with the programming *strategies* preferred by the *programmer*.

The program is written in a programming language that belongs to a programming paradigm and made of perceptual code and information types. Perceptual code and programming paradigm affect the information types displayed by the program in many ways – highlighting or obscuring it. Because of this, the programmer's performance on tasks can be affected, depending on whether there is a match between the information that is highlighted and that is required by the tasks.

The information displayed by the program also affects the programmer's mental representation of the program, affecting his comprehension of the program. This in turn affects the other programming activities subsequently carried out by the programmer.

The model has been derived from previous empirical research, most of which studied textual programming languages. There is one concern, however, whether these findings are also applicable to visual programs because the Match-Mismatch phenomenon has not always been observed in visual programs as expected, as has always been the case for textual programs.

MoPP highlights two areas to be investigated further as they are not adequately researched in the literature. The two areas are: the effect of programming paradigm on program comprehension performance and the role of perceptual coding on enhancing program comprehension for visual programs.

On the paradigm front, in addition to the need to study the effect of programming paradigm on novices' performance, the literature has also indicated a possibility of paradigm preference among novices and hence, another issue for investigation.

In this chapter we have also explored design principles for diagrammatic languages and visual language design for visual programs. From this we have derived a Visual Language Matrix (VLM) for visual programs, thereby was able to generate a list of perceptual cues that can be used to enhance program *appearance*, which we define as referring to readability, legibility, comprehensibility, and maintainability of the program. The list provides perceptual cues in three modes: textual, spatial, and graphic. One of the perceptual cues least studied by previous research and which could have significant effect on program comprehension is layout of visual programs. We subsequently suggest that a study on representation of program flow should be conducted and that the study should attempt to provide an answer to whether a directional representation makes any difference in tracing a diagrammatic program and whether program comprehension is affected by the traversal direction.

3. PROGRAMMING PARADIGM: AN EMPIRICAL STUDY

3.1 Introduction

In Chapter 2, we discussed and explored previous research on the effect of programming paradigm on program comprehension. There is a strong indication that novices' performance is affected by the programming paradigm of the language they use in two ways. Firstly, programming paradigm influences the mental representation of the program formed by novices. Secondly, the programmer will find the language hard to use when there is no cognitive fit between the preferred programming paradigm and the paradigm that the language they use is in. This chapter presents an experiment that studies the effect of programming paradigm on program comprehension performance of novices. It also provides some data that indicate a control flow preference among the novices who participated in the experiments carried out for this research.

In designing the experiment it is necessary that issues that would be interesting or that would confound the experiment be considered. The sections that follow first investigate various different issues that have to be taken into consideration during the experimental design, followed by the description of the experiment and discussions of its findings. In the last section, we summarise what has been learned from the experiment and what other questions need to be answered.

3.2 Experimental design issues

3.2.1 Methodology

As we have already discussed in Chapter 2, evidence of programs as information displays comes from two main lines of research that look at mental representation of programs and that looks at programs as information displays. The former usually involves studying the effect that different programming paradigms have on programmers' performance [see, for example, Corritore & Wiedenbeck (1999)]. Whilst this line of research has been quite comprehensive, little has been studied of the effect of programming paradigm taking the latter approach: programs as information displays. Gilmore & Green (1984) found that the performance of backward tracing was better than that of forward tracing for a declarative style and that the opposite was true for a procedural style of the same program. This resulted in the 'Match-Mismatch' hypothesis (see Section 2.3.1). However, it did not give a clear picture of the effect of programming paradigm on overall performance. What is needed is more evidence from this line of research to support the findings by the mental representation researchers. Therefore, in this research, we conduct an experiment (presented in this chapter) to provide this evidence.

The methodology widely used by existing research (for example, Sime *et al.*, 1977a; Gilmore & Green, 1984; Sinha & Vessey, 1992) is quantitative and experimental. This present study adopts their methodology because, firstly, by employing the same methodology as previous researchers, our results can be compared with those of the existing research for triangulation. Secondly, controlled experiments facilitate comparison and are suitable when hypotheses can be formed. The aim of this present study is to compare novices' performance between programming paradigms and between program modalities and to test the hypotheses that are formed from our literature review in Chapter 2.

Traditionally, participants were required to do forward and backward tracing of some programs. Forward and backward response time performance was recorded separately and compared to provide evidence supporting the 'Match-Mismatch' hypothesis. We argue, however, that as we are comparing two different notations, both forward and backward performance should be taken into account when calculating the overall performance. This is because performance depends on tasks (forward or backward tracing) and on the programming language (notation) used (Gilmore & Green, 1984).

3.2.2 Choice of paradigm

We decided to compare the overall performance in tracing programs between the two most commonly used paradigms in visual programming: the control flow and the data flow. We chose control flow because it had been commonly used in the flowchart and the structured flowchart era. Furthermore, there seems to be a control flow preference among novices (discussed in Section 2.4.2). We chose the data flow because major commercial visual programming languages such as Prograph, LabView, and HPVee are data flow languages.

3.2.3 Traversal direction

Traversal direction of diagrams may affect comprehension performance as discussed in Section 2.5.4. If this speculation is correct it is desirable to do the test with more than one traversal direction. However, it is not viable to try every possible traversal direction in a study. A few selected ones should suffice. If the effect of programming paradigm can be found, we should also see that the effect persists regardless of traversal direction used.

3.2.4 Program modality

There are conflicting research results in the literature whether text is better than diagrams or vice versa as summarised here in Tables 3.1 and 3.2. Green *et al*'s (1991) and Moher *et al.*'s (1993) studies show a clear-cut superiority of textual programs over visual programs. However, they used a micro-language called Nest-INE which has been shown to give better performance compared to conventional style languages (Sime *et al.*, 1979). An example of a Nest-INE program can be found in Figure 2.2 in Chapter 2. We feel that a fair comparison should be made and hence, a conventional textual program should be used in our experiment.

Table 3.1 Evidence favouring diagrams over text

Authors	Representations compared	Findings
Wright & Reid (1973)	Prose, sentences, decision tree, and decision tables	Most errors were made with prose. Decision tree performed best, particularly for complex problems.
Blaiwes (1974)	Sentences and flowchart format	Flowchart was more accurate than short sentences for use as instructions for difficult problems.
Kammann (1975)	Prose and flowchart format	Flowchart was better than prose for use as instructions.
Fitter & Green (1979)	Backus-Naur form (BNF) and syntax diagrams	Syntax diagrams gave better speed performance for task that required tracing through the grammars, but not for that requiring knowledge of the structure of the grammar.
Brooke and Duncan (1980)	Flowcharts, Nassi-Shneiderman diagram, If-then-else listing , and If-branch to label listing	Diagrams were more useful than the listings for debugging tasks that demanded tracing of execution path.
Vessey & Weber (1986)	Structured English, decision trees, and decision tables.	Decision trees outperformed structured English and decision tables in representing conditional logic.
Scanlan (1989)	Pseudocode and structured flowcharts	Flowcharts outperformed pseudocode regardless of program size. Flowchart superiority increased as problem complexity increased.
Anjaneyulu & Anderson (1992)	DRLP and LISP	The advantage of DRLP over LISP was its potential to eliminate the certain kinds of error (DRLP programs are data flow graphs.)
Cunniff *et al.* (1989)	FPL and Pascal	FPL programming bugs were compared with Pascal bugs. • FPL bugs were much fewer than Pascal bugs. • FPL was superior to Pascal for the absence of syntax-related bugs and of bugs relating to misplacing the code. (FPL programs are executable structured flowcharts.)
Catarci & Santucci (1995)	QBD and SQL	QBD was superior to SQL in both time and accuracy performance for all user levels: naïve, intermediate, and expert. (QBD is a diagrammatic query language.)
Glinert & Tanimoto (1990)	User satisfaction for Pict system	98.2% of their participants liked Pict flowcharts. (Pict programs are executable flowcharts.)

Table 3.2 Evidence favouring text over diagrams

Authors	Representations compared	Findings
Brooke and Duncan (1980)	Flowchart and list of short sentences	Flowchart did not improve fault identification but it appeared to facilitate tracing conditional logic.
Gilmore & Smith (1984)	Flowchart, program listing, and Bowles diagram	Flowchart did not improve debugging performance but the authors concluded that flowchart usefulness depended on the nature of task and individual programmer characteristics.
Ramsey *et al.* (1983)	PDL and flowchart	PDL was superior to flowcharts in program design and flowcharts benefited from spatial arrangement.
Curtis *et al.* (1989)	Nine combinations of symbology and spatial arrangement.	The combination of constrained language and sequential arrangement, which is equivalent to PDL, was the best performance overall. Branching highlights control flow information better than other arrangements. (Tasks: coding, comprehension, modification, and debugging; Symbology: natural language, constrained language, and ideogram; Spatial arrangement: sequential, branching, and hierarchical)
Green *et al.* (1991)	LabVIEW, Nest-INE textual notation (Sime *et al.*, 1977b) LabVIEW, Do-If textual notation (Green *et al.*, 1991)	Graphical programs took longer time than textual ones.
Moher *et al.* (1993)	Petri net, Nest-INE textual notation (Sime *et al.*, 1977b) Petri net, Do-If textual notation (Green *et al.*, 1991)	Petri net programs did not outperform the textual programs and some were much worse than the textual programs.
Halewood & Woodward (1993)	User satisfaction for GRIPSE	Neutral satisfaction. Users experienced difficulty in zooming at nesting and in manipulating the Nassi-Shneiderman charts. GRIPSE (Graphical Integrated Programming Support Environment) programs are NS charts.

3.3 General description of the experiment

3.3.1 Objectives

The objectives of this experiment are the following:

1. To study the effect of programming paradigm on comprehensibility of visual programs: whether a control flow program would be better than a data flow program, or vice versa.
2. To see whether visual programs would do better or worse than textual ones.
3. To study the effect of traversal direction on visual programs.

To achieve the above objectives, we compare programmers' performance of the followings:

1. Visual programs vs conventional procedural textual program.
2. Control flow vs data flow visual programs.
3. Three traversal directions: Top-Down; Hierarchical-Nested; and Free-Style.

3.3.2 Description of traversal directions in the visual programs

The three traversal directions used in this experiment are described below. The schematic representations for the control flow programs are given in Figure 3.1. Arrows represent the direction of flow. The rectangular boxes in the diagrams represent operations and the diamond shapes represent decision points.

Top-Down (TD)

The program is traversed from top to bottom of the screen. At a decision point, the two Yes and No arms branch to left or right.

Hierarchical-Nested (HN)

The program is read from the top leftmost primitive down the vertical line. When a small circle is reached, traversing takes the branch on the right. When all possible branches have been traversed, '*fall back*' (Green, 1982) occurs. That is, one returns to the point before branching off, i.e. the small circle. Then traversing resumes in a downward direction until another small circle is reached and branching off takes place as mentioned before. This process repeats until the horizontal line at the bottom of the vertical line is reached. If the vertical line is not the leftmost one, '*fall back*' occurs. Otherwise traversing is complete.

Free-Style (FS)

The program is traversed by following arrows. Arrow was used because the traversing is not restricted in any particular direction, readers can become confused if a line is used. This would confound the data. In this experiment graphical primitives were placed as randomly as possible, but within an acceptable degree of layout organisation.

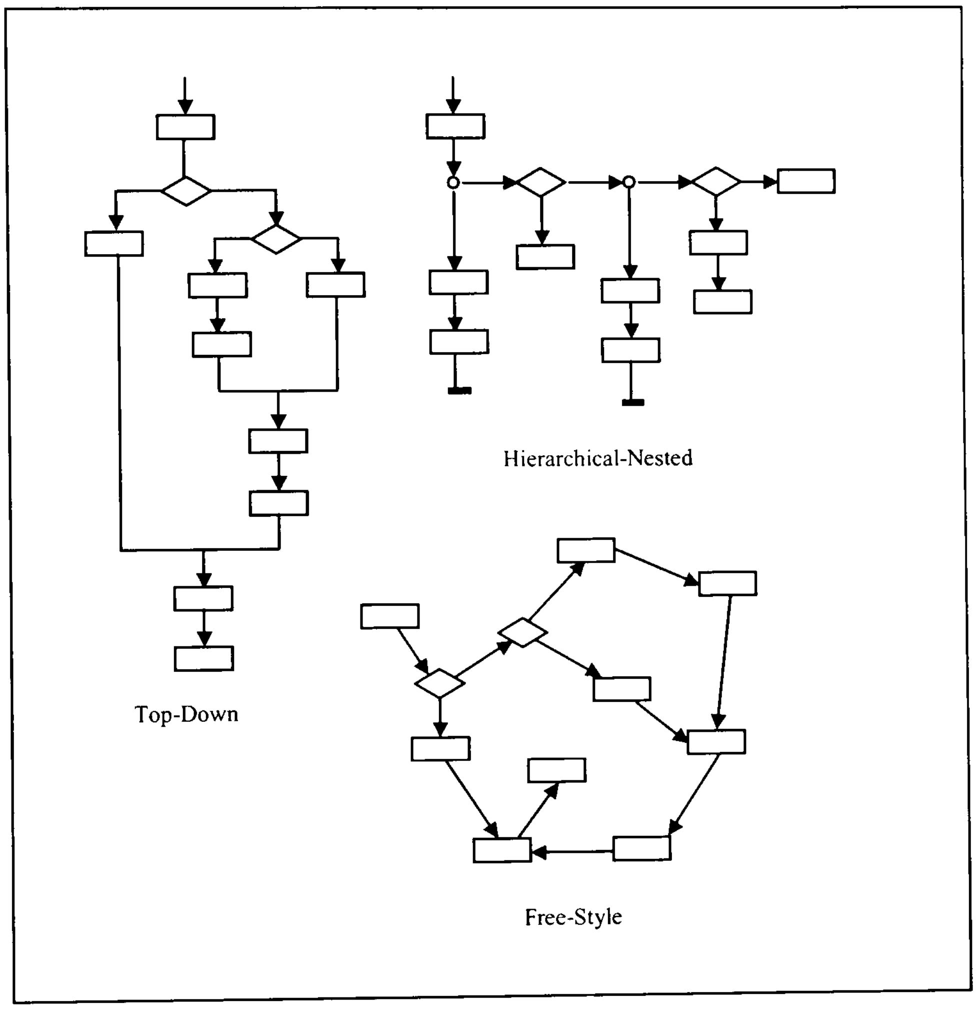


Figure 3.1 Traversal directions used in the experiment

3.3.3 Hypotheses

The textual program (Figure 3.2) is very similar to a conventional program listing. Even though indentation and white space were used, we anticipated that the number of If, Else, End If, and End loop would make tracing the program difficult. With the visual programs there was no redundant representation that would clutter or confuse readers. Layout organisation varied in degree of clarity so performance in some visual programs might suffer. Nevertheless we expected that the textual program would be very hard to trace.

```
If S = '*Pretty*' then
   Loop begins for Times = 1 to 2
      If S = '*Sad*' then
         Print 'Shout'
      Else
         Print 'Goal'
      End if
   End loop
Else
   If S = '*Funny*' then
      Print 'Nod'
   Else
      If S = '*Sad*' then
         Print 'Goal'
      End If
   End If
End If
```

Figure 3.2 A part of the textual program

Hypothesis 1. We expected that Top-Down would outperform text. Top-Down resembles family trees and organisational charts so we expected participants to be familiar with it. Omerod *et al.* (1986) showed that diagrammatic representation of family relationships outperformed text, but for unfamiliar relationships the advantage of diagrams over text was reduced. However, to answer comprehension questions the participants in their study had to examine two diagrams or two lists. In our experiment to answer a question participants examined only one representation. Moreover, the branching arms go in opposite direction from the decision point, and always either to the left or to the right, not in any random direction. Therefore, the advantage of the diagram's layout organisation and branching over sequential text should be more apparent. We expected that this style would outperform text.

Hypothesis 2. We speculated that Hierarchical-Nested and text performance might not differ much. Hierarchical-Nested requires '*fall back*' (Green, 1982) which would increase the participants' mental load and may cause them to forget to return to where they left off or to confuse them. Branching would be an advantage over text but we did not know the net effect of '*fall back*' and branching.

Hypothesis 3. Diagrams were larger than the screen space available and scroll bars were provided. The problem that we anticipated with the Free-Style diagram is finding where to start if the starting graphical primitive could not be seen when the diagram first appeared. This could make comparison unfair because the starting point of the textual program and the other two styles could be recognised when they were seen the first time. The starting graphical primitive of the Free-Style diagram was, therefore, brought to the centre of the screen when it was first shown. The random placement of graphical primitives and diagram size would be a disadvantage but branching would be an advantage over text. However, since arrows were used and there was no '*fall back*', we speculated that this style might perform better than text.

Hypothesis 4. We expected that data flow performance would be much poorer than control flow performance. Data flow programs were generally larger than their control flow counterparts. A limited scrollable screen space would make tracing difficult and would increase mental load on the participants.

Hypothesis 5. Based on the Gilmore & Green's (1984) Match-Mismatch hypothesis (discussed in Section 2.3.1) we expected that forward questions would be easier than backward questions for a sequential program (the control flow program) and that backward questions would be easier for a circumstantial program (the data flow program).

3.3.4 Method

Design

The experiment was a within-subjects design and consisted of control flow and data flow sub-experiments. All participants performed one control flow sub-experiment and one data flow sub-experiment. In each sub-experiment for each of the four programs presented, three visual programs and one textual program, participants were asked to answer forward and backward questions.

Participants

Twenty-two undergraduate students at Brunel University participated in this experiment at the end of their first year. Of all participants, eighteen were Computer Science students, three were Mathematics students and one was an Engineering student. All were paid £35 for participating in both morning and afternoon sessions. Tea, coffee, and snacks were provided during breaks at the departmental staff coffee area. Lunch was not provided to participants. They were given an hour lunch break between the two sessions.

Materials

Programs

The programs (see their textual version in Appendix A-1) consisted of conditional structures and simple loops. Seven programs were used: one textual program and its corresponding visual representations in Top-Down, Hierarchical-Nested, and Free-Style each in both the control flow and in the data flow paradigm. The textual program was a small conventional style program that matches a string, S, for adjectives (Bad, Pretty, Sad, etc.) and prints verbs (Wink, Shout, Nod, etc.). The program content was designed to be meaningless so that participants would not remember the answers or give answers based on their experience, which would confound experimental data. Appendix A-2 illustrates three control-flow and three data-flow sample programs in the three traversal directions used. Since the actual complete program is large, for demonstration purposes and ease of understanding, the sample programs given in the Appendix are representations of the same small fragment of a full program that was not used in this experiment. Because it is not possible to fit a data-flow program (or a part of it) into an A4-size page in this thesis in such a way that is comprehensible to readers, only the control-flow programs are given in full in Appendix A-3.

In the control flow sub-experiment, control flow programs were used. In the data flow sub-experiment, data flow programs were used. The same textual program was used in both sub-experiments. The textual program should not be affected significantly by paradigm difference but it was included for completeness (discussed later in Section 3.5). A part of the textual program is given in Figure 3.2 whilst the full program can be found in Appendix A-1.

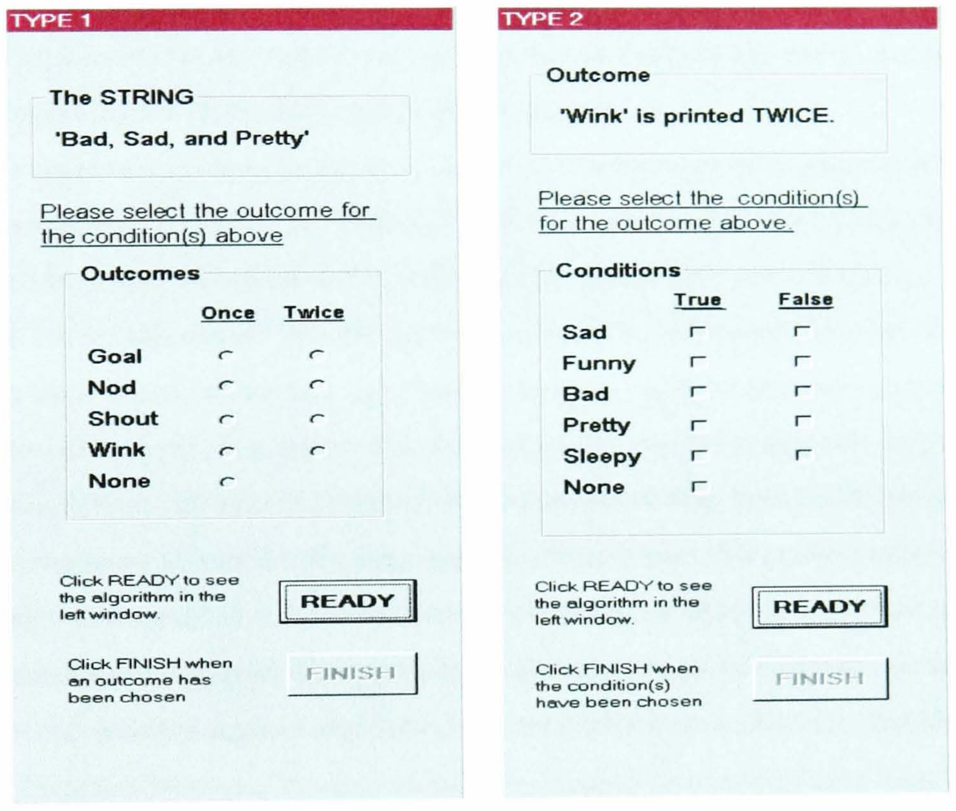
In control flow programs, arrows represented flow of control. In data flow programs, arrows represented flow of data. The data flow programs used the token model. The conditional construct in the data flow programs used a selector and a distributor as described by Shu (1992) with slight modifications. In this experiment, representations of a distributor and of a selector were a hexagon and a capsule-like shape, respectively (Appendix A-4). The distributor has two inputs and two outputs. It uses one of the inputs to determine which of the two output arcs to send the incoming data token to. The selector has three inputs and one output. It uses the horizontal input to determine which vertical input to pass to the output. A little square was used as a connector from a distributor to a selector to reduce the number of lines in the data flow diagrams. Both used the same iteration construct, which encapsulated the iterative process. The iterative process was represented by flow of control, test nodes, and action nodes in control flow programs. In data flow programs, the iterative process was represented by flow of data, test nodes, and function nodes.

Tasks

Each participant answered four forward and four backward questions similar to those used by Green *et al.* (1991). Forward questions are questions that give conditions and ask for the outcomes. Backward questions give the outcomes and ask for the conditions. Examples of the questions are given in Figure 3.3.

Application program

The experiment was administered online, using a 17-inch monitor, 1024 x 768 pixels screen resolution. The program was written in Visual Basic by the author of this thesis. It recorded the response time and answers from the participants.


TYPE 1

The STRING

'Bad, Sad, and Pretty'

Please select the outcome for the condition(s) above

Outcomes

	Once	Twice
Goal	○	○
Nod	○	○
Shout	○	○
Wink	○	○
None	○	

Click READY to see the algorithm in the left window.

READY

Click FINISH when an outcome has been chosen

FINISH

TYPE 2

Outcome

'Wink' is printed TWICE.

Please select the condition(s) for the outcome above.

Conditions

	True	False
Sad	☐	☐
Funny	☐	☐
Bad	☐	☐
Pretty	☐	☐
Sleepy	☐	☐
None	☐	

Click READY to see the algorithm in the left window.

READY

Click FINISH when the condition(s) have been chosen

FINISH

Figure 3.3 Forward question (left) and backward question (right)

Procedure

Participants were given a forty-five-minute training session per paradigm. Parts of the training notes can be found in Appendices A-5 and A-6. A pre-test questionnaire was given at the beginning of the tutorial session which asked participants about their programming experience. This questionnaire can be found in Appendix A-7. Half of the participants were taught the control flow paradigm first while the other half were taught the data flow paradigm first. The purpose was to ensure that everyone knew how to read and understand the programs in all representations. No assumption was made that participants were conversant with the textual program mode. All working examples during the training sessions were based on the same program, which was different from those in the experiment proper and the online practice. Sample questions to test their understanding contained both forward and backward questions.

After training, participants took the test for the paradigm they were taught. Later in the second session participants were taught the other paradigm. The same procedure as the previous session was followed, and the test that participants took was for the paradigm they newly learned.

All participants were given an online practice session first. A sample program that mimicked the real test was run and the participants went through the whole procedure at their own pace. Participants could repeat the practice if they wished. At the end of the practice test, the program informed the participants of their scores.

In the experiment proper participants answered four forward questions and four backward questions, one pair of forward and backward questions for each program mode. The order of diagrams, questions, and question types seen by each participant was randomised. Participants never saw the same diagram or the same question type on two consecutive trials. The first diagram and first question type of the series that each participant saw was also randomised. The screen was divided into two sections, diagram and question-answer sections. First, the question-answer section appeared with a graphical image irrelevant to the problem task on the other section of the screen. The participant clicked the button *Ready* on the question-answer section when he/she felt ready to start. A diagram along with scroll bars appeared. The participant worked through the diagram and clicked the answer(s) in the question-answer section and the *Finish* button when he/she finished. During this period, response time was recorded along with the final answers and question details. The whole process was repeated until all eight questions were answered. Before the program ended, the program informed the participant of the total marks he/she achieved.

3.4 Results

The mean total score achieved was 6.24 for the control flow experiment and 5.55 for the data flow experiment. There were 21 and 22 participants in the control-flow and data-flow experiments, respectively. The data-flow data for the participant who did not take part in the control-flow experiment was discarded. The mean of the total response time taken to answer each pair of questions (forward and backward questions) and the mean of sum score of the two question types (one mark per question) are given in Tables 3.3 and 3.4, respectively. Data analyses for control flow, data flow, and paradigm comparisons are subsequently carried out. Note that line graphs are sometimes used for readability purpose.

Table 3.3 Mean response time to answer both question types by each participant

Program Mode	Control flow experiment (N = 21)		Data flow experiment (N = 21)	
	RT (s)	SD	RT (s)	SD
Graphics (Top-Down)	79	49	110	54
Graphics (Hierarchical-Nested)	78	37	160	84
Graphics (Free-Style)	86	31	177	107
Text	104	36	147	93

Table 3.4 Mean score of both question types achieved by each participant

Program Mode	Control flow experiment (N = 21)		Data flow experiment (N = 21)	
	Score	SD	Score	SD
Graphics (Top-Down)	1.67	.58	1.62	0.59
Graphics (Hierarchical-Nested)	1.67	.58	1.57	0.60
Graphics (Free-Style)	1.76	.54	1.48	0.68
Text	1.14	.65	1.10	0.62

3.4.1 Control flow experiment

Data analyses are conducted on response time and accuracy performance analyses as described below. The ANOVA, t-test and McNemar test statistics for the control flow experiment are given in Table 3.5.

Response Time Analysis

A two-factor, repeated measures ANOVA was performed. The two factors were program mode (Top-Down, Hierarchical-Nested, Free-Style, and Text) and question type (Forward and Backward). The dependent variable was the response time for the question. The ANOVA revealed a main effect of program mode (see Figure 3.4 and Table 3.5). The degrees of freedom have been adjusted with the Huynh-Feldt epsilon to correct for violation of sphericity assumption. No main effect of question type or interaction was found.

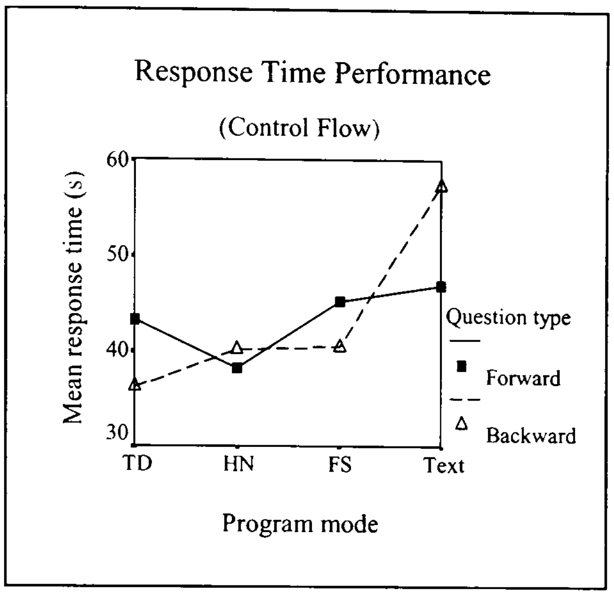


Figure 3.4 Control flow: response time performance

Planned comparisons of the performance of both question types combined for two graphics-text pairs for Top-Down and Free-Style were made. The statistics revealed marginal difference for the two graphics-text pairs. Unplanned comparison for the Hierarchical-Nested and Text pair was made; the Bonferroni *p* value of 0.02 was used. The t-test revealed a significant difference between Hierarchical-Nested and Text.

As for the effect of question type, even though the ANOVA did not find a main effect of question type on these four programs, findings in the literature (as discussed in Chapter 2) led us to speculate regarding question type effect on textual programs. Pairwise comparison between the two question types for the text program was made and a significant difference between forward and backward questions for the textual program was revealed.

Accuracy Analysis

A one-factor repeated measures ANOVA was performed. The independent variable was program mode (four levels: Top Down, Hierarchical-Nested, Free Style, and Text). The dependent variable was percent correct responses for both forward and backward questions. The ANOVA result revealed a strong effect of program mode, $F(3,60) = 5.62, p < 0.01$ (see Figure 3.5 and Table 3.5).

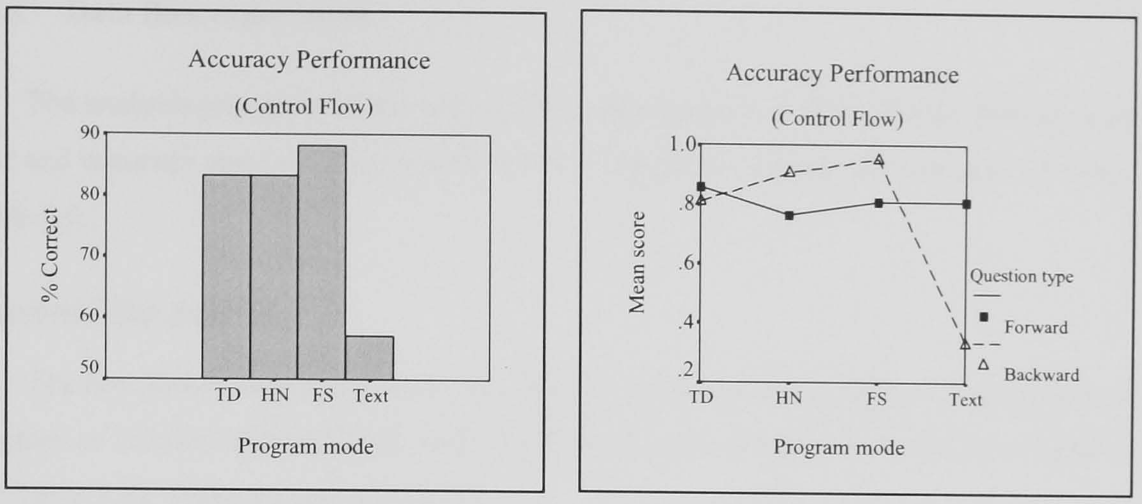


Figure 3.5 Control flow: Accuracy performance

Following the response time analysis above, pairwise comparisons between the graphics-text pairs were carried out for the performance of both question types combined. Planned comparisons revealed a significant difference for Top-Down and Free-Style. Unplanned comparison between the Hierarchical-Nested and Text pair also revealed a significant difference; the Bonferroni *p* value used was 0.02.

As for the effect of question type, McNemar tests for dichotomous nominal data analysis for score obtained in each question type was carried out for each program mode separately. There was no main effect of question type in any of the three visual program mode. However, a significant difference between forward and backward questions was found for the textual program.

Table 3.5 Control Flow: ANOVA and t-test statistics

Factor	Response time	Accuracy
Program mode	F (2.21, 44.16) = 3.18, $p < 0.05$	F(3,60) = 5.62, $p < 0.01$
Question type	F(1, 20) = 0.002, ns	-
	Interaction: F(3,60) = 2.17, ns	-
Significant difference		
Graphics-Text	TD: $t(20) = 1.86$, $p = 0.08$	TD: $t(20) = 2.75$, $p = 0.01$
	HN: $t(20) = 2.62$, $p = 0.02$	HN: $t(20) = 2.95$, $p = 0.01$
	FS: $t(20) = 1.98$, $p = 0.06$	FS: $t(20) = 3.83$, $p = 0.001$
Forward-Backward	Text: $t(20) = 2.12$, $p= 0.05$	Text: McNemar's, p = 0.006

3.4.2 Data flow experiment

The analysis procedure in the control flow experiment was followed for both response time and accuracy analyses. The ANOVA, t-test and McNemar test statistics are given in Table 3.6.

Response Time Analysis

The two-factor, repeated measures ANOVA revealed a main effect of program mode (degrees of freedom adjusted) but no main effect of question type was found (see Figure 3.6 and Table 3.6). There was no interaction.

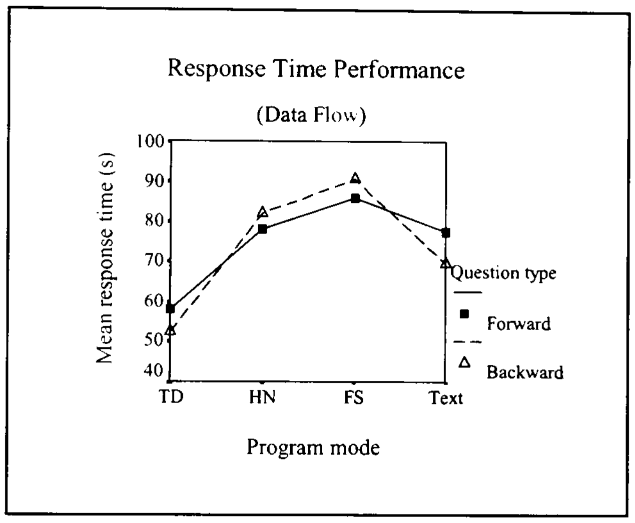


Figure 3.6 Data flow: Response time performance

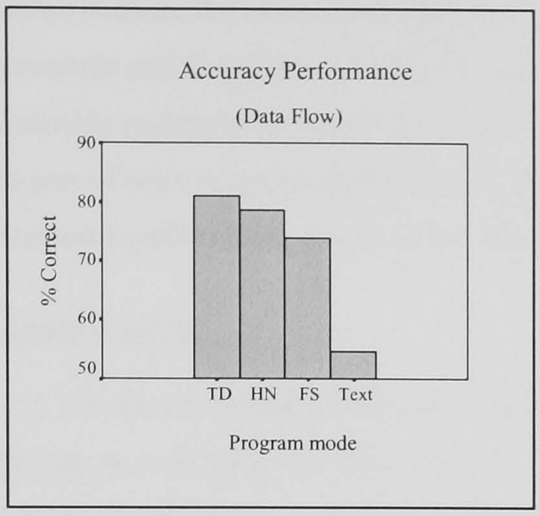
Planned comparison of the graphics-text pairs for Top-Down and Free-Style revealed a significant difference between the Top-Down and Text only. Unplanned comparison was made for the Hierarchical-Nested and Text pair; no significant difference was found. The t-test statistics revealed no significant difference between the two question types for Text.

Accuracy Analysis

Following the control flow accuracy analysis, the ANOVA revealed a strong main effect of program mode, F(3,60) = 5.4, $p < 0.01$ (see Figure 3.7 and Table 3.6).

The t-tests also revealed that all visual programs outperformed Text.

As for the effect of question type, McNemar tests for dichotomous nominal data analysis for score obtained in each question type was carried out for each program mode separately. There was no main effect of question type in any of the four programs.



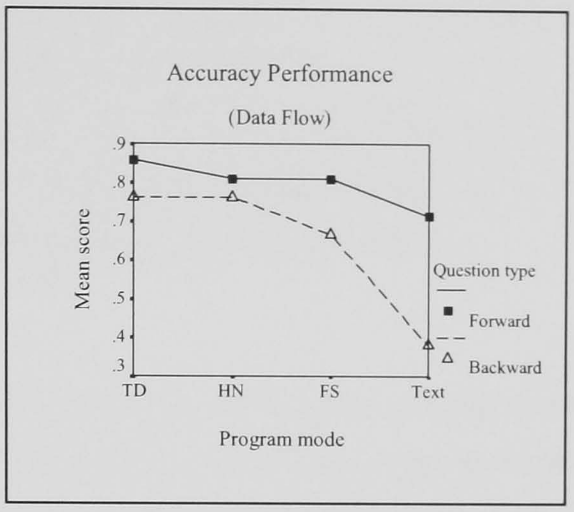


Figure 3.7 Data flow: Accuracy performance

Table 3.6 Data Flow: ANOVA and t-test statistics

Factor	Response time	Accuracy
Program mode	F(1.87, 37.41) = 4.48, $p < 0.02$	F(3,60) = 5.4, $p < 0.01$
Question type	F(1, 20) = 0.05, ns	-
	Interaction: F(3,60) = 0.29, ns	-
Significant difference		
Graphics- Text	TD: $t(20) = 2.16$, $p = 0.043$	TD: $t(20) = 3.20$, $p = 0.004$ HN: $t(20) = 4.26$, $p = 0.0005$ FS: $t(20) = 2.96$, $p = 0.008$
Forward-Backward	Not significant	Not significant

3.4.3 Paradigm analysis

Response time and accuracy analyses are carried out as described below. ANOVA and t-test statistics are given in Table 3.7. Figure 3.8 shows the effect of programming paradigm on the visual programs.

Response Time Analysis

Control flow/data flow participants were matched. A two-factor, repeated measures ANOVA was performed. The two factors were program mode and paradigm. The dependent variable was the sum of response time for both types of questions. The ANOVA revealed the main effects of program mode (degrees of freedom adjusted) and of paradigm. However, there was an interaction, program mode * paradigm (degrees of freedom adjusted). This was expected because the same textual program was used in both control flow and data flow sub-experiments and therefore should not be affected by paradigm difference.

Pairwise comparisons were then made between the control flow and the data flow programs of each program mode. The t-tests revealed that the control flow programs performed significantly better than the data flow programs for all program modes.

Accuracy Analysis

A two-factor, repeated measures ANOVA analysis was performed on accuracy data the same way as in the response time analysis. The dependent variable was the percent correct responses for both types of questions. The ANOVA revealed a strong main effect of program mode, but there was no main effect of paradigm. Nor was there an interaction.

Table 3.7 Paradigm comparison: ANOVA and t-test statistics

Factor	Response time	Accuracy
Program mode Paradigm	F(1.64, 32.85) = 4.98, $p < 0.05$ F(1, 20) = 25.6, $p < 0.001$ Interaction: F(2.6, 51.93) = 3.56, $p < 0.05$	F(3,60) = 11.75, $p < 0.001$ F(1,20) = 2.02, ns Interaction: F(3,60) = 0.747, ns
Significant difference		
Control flow vs Data flow	TD: $t(20) = 2.61, p = 0.02$ HN: $t(20) = 4.78, p = 0.0005$ FS: $t(20) = 4.39, p = 0.0005$	

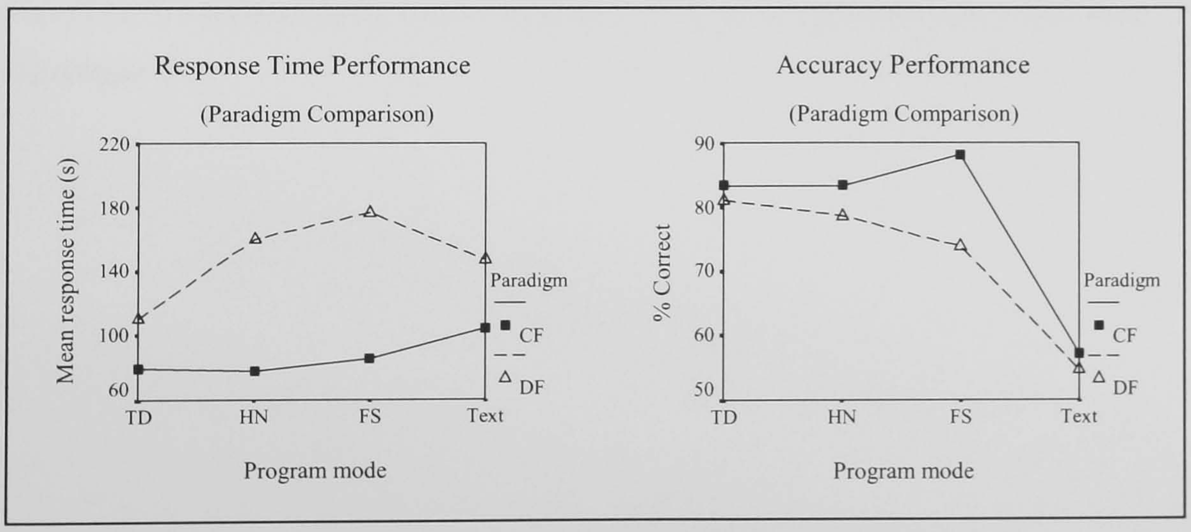


Figure 3.8 Paradigm effect on accuracy and response time performance

3.5 Discussion

Participants' accuracy performance showed the superiority of the visual programs over text in almost all cases. From the summary of the findings in this experiment tabulated in Table 3.8, the hypotheses formed in Section 3.3.3 are discussed below.

Hypotheses 1 and 3 were supported for the control flow programs. In terms of accuracy performance, Top-Down and Free-Style outperformed text in both paradigms.

Hypothesis 2 was not supported. We speculated that '*fall back*' would be so difficult that Hierarchical-Nested would be outperformed by the textual program. Instead, we found that it outperformed text in both the response time and the accuracy performance except for the data flow programs.

Hypothesis 4 was supported in terms of response time only. The time taken to finish the task for the data flow programs was much longer than for the control flow programs. However, there was no paradigm effect in terms of accuracy. Programming paradigm only affected the response time performance in the visual programs.

Hypothesis 5 was supported for Text only. The effect of question type was not significant in any of the visual program. The textual program was written in a control flow language. The fact that we did not also obtain a significant effect of question type in the data flow sub-experiment might be because the problems that participants experienced with the data flow visual programs had affected their performance on the textual programs also. This is supported by the data in Table 3.3 and 3.4 showing poorer performance and higher standard deviation of the data-flow sub-experiment (response time = 147 s and SD = 93) for the textual programs than those of the control-flow one (response time = 104 s and SD = 36). Therefore, we maintain that it is reasonable to use the results from the control flow sub-experiment alone.

Table 3.8 Summary of findings

Factor	Finding		Response time		Accuracy	
			CF	DF	CF	DF
Program Mode	ANOVA main effect		yes	yes	yes	yes
	Significant difference:	TD - Text	marginal	yes	yes	yes
		HN - Text	yes	no	yes	yes
		FS - Text	marginal	no	yes	yes
Question Type	ANOVA main effect		no	no	-	-
	Significant difference:	TD	-	-	-	no
		HN	-	-	-	no
		FS	-	-	-	no
		Text	yes	no	yes	no
Paradigm	ANOVA main effect		yes		no	
	Significant difference:	TD	yes		-	
		HN	yes		-	
		FS	yes		-	

SUMMARY

Research questions:	Response Time		Accuracy	
	CF	DF	CF	DF
Is graphics better than text?	yes	inconclusive (TD-Text only)	yes	yes
Effect of question type	Text only	none	Text only	none
Effect of paradigm	The visual programs only		none	

(TD = Top-Down; HN = Hierarchical-Nested; FS = Free style, CF = Control flow, DF =Data flow)

3.5.1 Paradigm effect on response time performance

This experiment revealed significant paradigm effect on response time. Figure 3.8 shows this effect across all traversal directions used. Nevertheless, this effect was not observed with accuracy performance and therefore it needs an explanation.

There are two differences among the three visual programs: traversal direction and diagram size. These two factors affect each other and are difficult to control simultaneously. We wanted to see whether the effect of paradigm, if it exists, would persist across traversal directions. Despite our attempt to control factors possibly confounding the experiment, the data flow programs inevitably required larger diagrams than their counterpart control flow programs and therefore scrolling was inevitable. The difference in response time might have been due to the diagram size as scrolling adds extra time to searching and may have increased the demand on working memory. As participants scrolled for new information the old information becomes invisible on the screen and has to be held in the working memory waiting to be processed. However, the amount and the time that information can be held in working memory are limited. By the time all the information required for the answer was accessed the old information would have been lost and hence had to be accessed again.

However, what we want to establish is whether novices find control flow paradigm easier, as indicated by the literature. The effect on response time does not seem to be due to how easy or difficult the representations in different paradigms are. Therefore, we should now look at the effect of programming paradigm on accuracy performance because it relates more directly to how easy or difficult the notation is to novices. If the notation is difficult, accuracy performance should be poor.

3.5.2 Paradigm effect on accuracy performance

It was surprising that there was no paradigm effect on the accuracy performance. Participants were first year students and had no experience with data flow programming languages. Moreover, the programs used here strictly followed the data flow model (i.e. the distributor and the selector were not omitted, hence there were more lines to confuse readers). Yet, the accuracy performance was not significantly affected by paradigm difference. One explanation may be that these students were at the end of their second semester and had just learned to use entity relationship and data flow diagrams in an Information Systems module. So learning a data flow language may not be as difficult as we would expect novices and hence no statistically significant difference could be observed.

However, if the two paradigms are equally easy or difficult, there should not be any difference whether which paradigm was learned first. We then investigated the data of participants who were taught control flow first and data flow first separately. Their total accuracy performance was plotted in Figure 3.9. The graph shows that the group that was taught control flow first could learn and perform equally well with the data flow program (79% in both programs). The group that learned data flow first did better with the control flow program than with the data flow program (77% in control flow as opposed to 66% in data flow). So regardless of the first paradigm taught, both groups performed well with the control flow program. This is similar to the case of learning iteration and recursion, where it has been found that novices learned recursion more successfully if they learned iteration first, i.e. that they possess an adequate mental model for learning recursion (Kessler & Anderson, 1989). The programming problem used in this experiment emphasises control flow concept, one of the major difficulties in learning programming. Control flow is a programming concept that exists and needs to be mastered regardless of programming paradigm. The result suggests that learning a control flow language first provides novices with an appropriate mental model for control flow programming concepts enabling them to handle a data flow language more readily. However, why would one need to learn iteration first before recursion or control flow language first before data flow language? A possible answer is 'cognitive fit' as suggested by the second maxim of information representation. Therefore, the data in Figure 3.9 may be an indication that there is a control flow preference among the students. Nevertheless, this is only an observation that awaits more empirical data.

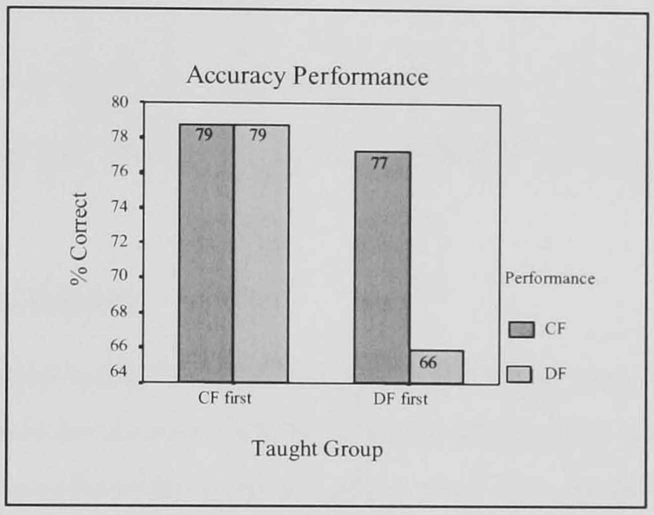


Figure 3.9 Control flow and data flow accuracy performance of the two taught groups

3.5.3 Control flow bias

In addition to the above indication that control flow may be easier to learn than data flow programs, we have some evidence of a control flow bias among our participants from questionnaire data collected from them throughout this research. The questionnaire respondents were first year students, who were undertaking computer studies at three different universities in the UK at the time. Table 3.9 shows percentage of procedural languages, object-oriented languages, and declarative languages that they previously had some knowledge of (excluding the language that they were taking at the time). Of the 131 respondents, the languages that they previously knew are 74.1%, 21%, and 4.9% for procedural, object-oriented, and declarative languages, respectively. These figures indicate a very high proportion of control flow languages.

Table 3.9 Questionnaire data for novices' previous programming languages

University	N	No. of languages per person	Percentage of the languages previously known		
			Procedural	Object-oriented	Declarative
U. of London at Goldsmiths[1]	51	0.5	72.0	28.0	0.0
Westminster[1]	18	0.8	53.3	33.3	13.3
Brunel[2]	43	0.5	85.7	9.5	4.8
Brunel [3]	19	1.1	80.0	15.0	5.0
Overall	131	0.6	74.1	21.0	4.9

(1= 1st semester, year 1999; 2 = 1st semester, year 2000; 3 = 2nd semester, year 2000)

3.5.4 Visual versus Textual

The visual programs outperformed the textual one in most cases. In terms of response time performance, this is not the case, particularly with the data flow programs. As we have discussed earlier, this may be affected by scrolling because the data flow programs were larger than the control flow and the textual programs. However, in accuracy term, the visual programs outperformed the textual program across all traversal direction in both paradigms. The result is clear. In this experiment, the visual programs are superior to their conventional textual counterpart for a small section of program with an emphasis on conditionals.

3.5.5 The 'Match-Mismatch' phenomenon in visual programs

Hypothesis 5 (that forward questions would be easier than backward questions for the control flow program and that backward questions would be easier for the data flow program) was not supported for the visual programs but for the textual program only. The effect of question type was not significant with visual programs. This result agrees with the literature that 'Match-Mismatch' phenomena have been found in textual programs but not in visual programs. This puts into doubt the applicability of research in psychology of programming to visual programming. Therefore this issue has yet to be investigated further.

3.5.6 The effect of traversal direction

The visual programs used in this experiment varied by traversal direction because we speculated that it might have an effect on performance. However, studying how traversal direction affects performance was not the aim of the study and therefore t-tests were not carried out for all possible test pairs to avoid Type II error in statistical analyses.

Performance data in Figure 3.10 indicate a possible traversal direction effect on response time performance of the data flow programs. We therefore carried out further data analyses below.

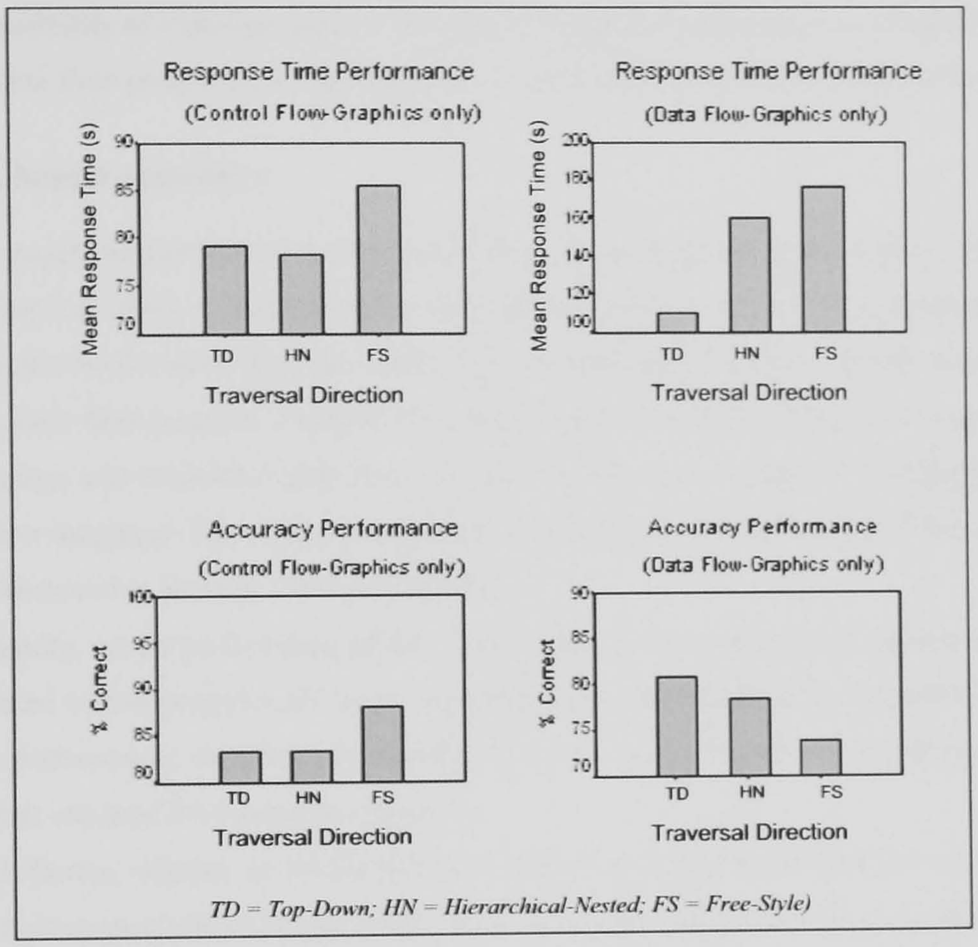


Figure 3.10 Performance on both question types for the visual programs

One-factor ANOVA analyses on both response time and accuracy for the control flow and the data flow visual programs was carried out. The independent variable was traversal direction and the dependent variable was the sum of forward and backward response time and percentage of correct responses, for the response time and accuracy analyses respectively. The ANOVA revealed no main effect of traversal direction on response time performance of the control flow programs, F(2, 40) = 0.520 (ns) and for accuracy performance of both the control flow and the data flow programs, F(2,40) = 0.241 (ns) and F(2,40) = 0.455 (ns), respectively. However, a main effect on response time performance for the data flow programs was found, F(1.532, 30.643) = 9.975, $p < 0.005$. The degrees of freedom have been adjusted with the Huynh-Feldt epsilon to correct for violation of sphericity assumption. Pairwise comparison revealed significant difference between Top-

Down and Hierarchical-Nested, $t(20) = 3.481$, $p=0.002$ and between Top-Down and Free-Style, $t(20) = 4.788$, $p=0.0005$.

Although we found a significant difference between the visual programs, we suggest that one be cautious in concluding that there is an effect of traversal direction on comprehensibility of visual programs. The reason is that the difference was obtained only with the data flow programs. As discussed above, this could be due to scrolling effect.

3.6 Chapter summary

The experiment presented in this chapter focused on programming paradigm issues. Three objectives of the experiment have been fulfilled. Firstly, on the role of programming paradigm, the results show that the control flow program seems to have a better 'cognitive fit' than a data flow program. Transfer from learning the control flow language to the data flow language was evidently easier than transfer from learning the data flow language to the control flow language. This adds yet another indication to a control flow preference among novices discussed in Section 2.4.2 in Chapter 2.

Secondly, on the performance of the visual programs in comparison to that of the conventional textual program, all three visual programs outperformed the textual one in accuracy performance, indicating the benefit of visual representations in enhancing the information required for tracing the programs.

And, finally, whether or not the observed performance is affected by traversal direction, the effect of traversal direction was observed across all three data flow visual programs but not in the control flow programs. This, we have discussed the reasons why it may have been due to scrolling as the data flow programs were larger than the control flow programs. Consequently, traversal direction seems to have no effect on tracing performance. Therefore, Green's (1982) armchair analysis, i.e. that '*fall back*' is difficult, is not supported by the results of this experiment.

In addition to the above findings serving the objectives of the experiment, the results of the experiment did not support the Match-Mismatch hypothesis in visual programs. This agrees well with Curtis, *et al.*'s (1989), Moher, *et al.*'s (1993), and Good's (1999) findings. The question whether the research in psychology of programming is applicable to visual programs thus still remains open.

4. REPRESENTATION OF PROGRAM FLOW

4.1 Introduction

One of the two areas identified by the Model of Programming Process (MoPP) derived in Chapter 2 to be explored in this thesis is perceptual coding. Perceptual coding has a role in either promoting or obscuring the information represented by a program. This role is expected to be more significant in visual programs than in textual programs where text usage is greatly reduced. This aspect of designing representations of programming objects and constructs must therefore be attended to, so that required information is highlighted or made less obscure. By doing so, comprehensibility could be improved. Upon our review in Chapter 2 of the literature on issues regarding perceptual coding, we summed up design principles for diagrammatic languages and also derived a Visual Language Matrix (VLM) for visual programs based on the existing VLM of textual documents proposed by Kostelnick & Roberts (1998). From the VLM of visual programs, we produced a list of perceptual cues that could be used to enhance the *appearance* (which we defined as referring to readability, legibility, maintainability, and comprehensibility) of visual programs. From the cues available, however, we identified that layout of visual programs and hence representation of program flow should be explored in order to answer the research questions raised during our review of the literature.

This chapter presents a series of empirical studies that investigate the effect of representation of program flow on novices' comprehension in visual programs. There are two aspects of program flow that we investigate in this chapter: directional representation and traversal direction. Despite the fact that the literature indicates possible effects of traversal direction on tracing programs (see Chapter 2 for the discussion), we found no effect of traversal direction in the experiment presented in Chapter 3. The lack of an effect of traversal direction on visual programs thus requires an explanation.

We conducted the experiments using the methodology traditionally employed by existing research in the literature for the reasons we have already as discussed in Section 3.2.1. The section that follows gives an overview of the work carried out, which consists of four experiments making up two major studies investigating directional representation and traversal direction. Each study is described and discussed in detail in subsequent sections. Findings of both studies are then summarised in the final section.

4.2 An overview of the conducted studies

Historically arrow, line, containment (boxes inside boxes), and juxtaposition (puzzle-like) have been used to represent program flow. The choice of representation can affect the performance of a user of such systems in following a sequence of actions, both in terms of time and of accuracy. However, there is little empirical evidence that would justify the use of one representation over another. The choice of representation is not necessarily governed by consideration of the user. Juxtaposition may be chosen for its economy of screen space, line for its bi-directional property, and arrow for its familiarity as a directional representation. A series of experiments presented here focus on two issues: the representation of direction of flow and traversal direction. The definition of and discussion about traversal direction can be found in Chapter 3.

Four experiments are presented in this chapter as follows:

1. Maze Study

This study focuses on the issue of visual representations for direction in general. It consists of two experiments: Maze Study 1 and Maze Study 2. The detail of the experiments is given in the next section.

2. Flow Study

This study focuses on representation of program flow, both the representation for direction and for traversing. It consists of two experiments: Flow Study 1 and Flow Study 2. We describe and discuss the detail of the experiments after the Maze Study.

In both studies, the first experiment acts as a pilot test that helps form a better design for the second one and for confirming experimental results. Maze Study 1 and Maze study 2 are presented together under Section 4.3 as they differ only in the order of the representations to which the participants were subjected. Flow Study 1 and Flow Study 2 have their own entire sessions (Section 4.4 and 4.5) as they are quite different in their designs.

4.3 The Maze Studies

4.3.1 Objective

The purpose of this study was to assess which of the three most commonly used flow representations: *Arrow*, *Line*, and *Juxtaposition* would be the best in both response time and accuracy performance.

4.3.2 General description

The experiments tested the ability of the participants to follow a direction. Commonly used diagramming techniques such as flowchart or structured flowcharts were not used in this study since each technique has its own inherent concepts that need to be learned and understood. Three 'maze' diagrams which differed by directional representations were used here (see an example of these diagrams 4.1). This representation is a route map representing all possible routes connecting *n* starting points (names of travellers) to *m* destination points (cities). It required participants only to follow a route/direction. This representation has been chosen because it is similar to diagrams used for bus routes, train networks, and underground maps and therefore should be familiar to the participants in the study. The maze consists of only three types of objects: starting points, destination points, and one directional notation for each maze to indicate paths or routes.

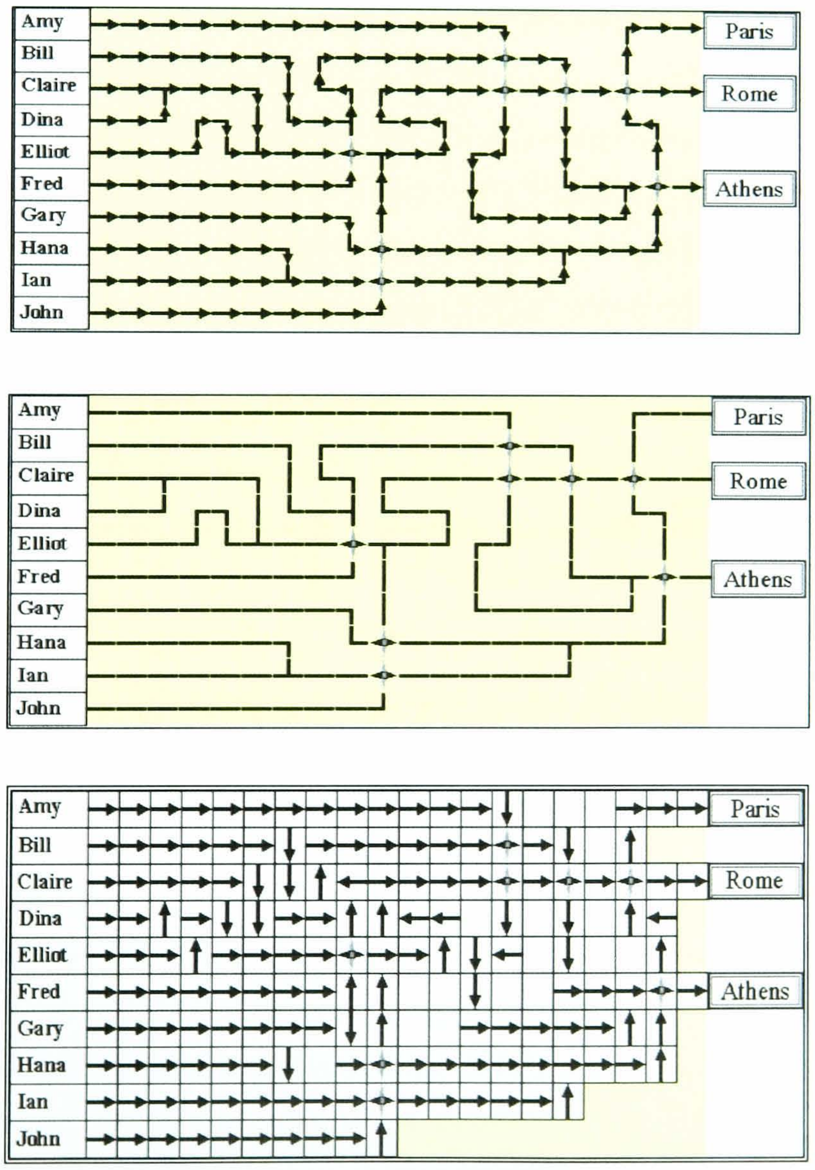


Figure 4.1 The three mazes: arrow (top); line (middle); and juxtaposition (bottom)

4.3.3 Hypotheses

The following hypotheses were formed for this experiment:

Hypothesis 1. Response time and accuracy performance would be affected by the choice of directional representation.

Hypothesis 2. Arrow would yield the best performance in both response time and accuracy performance.

4.3.4 Method: Maze Study 1

Design

The experiment is a mixed factorial design. Two groups of subjects were presented with three directional representations (*Arrow*, *Line*, *Juxtaposition*) and three trials per representation. It was not the intention to study the difference between subject groups. Nevertheless, because our volunteers were from different universities the experiment was a mixed factorial design so that difference between the two groups from different universities could also be investigated.

Participants

Participants consisted of two separate groups of first year Computer Science undergraduate students. Nineteen students participated in the experiment at the University of Westminster and eighteen participated at Goldsmiths College, University of London. Both groups were doing their first programming language courses in their first year: Visual Basic at Westminster and Pascal at Goldsmiths College.

Materials

We conducted two experimental sessions in a computer laboratory, one on each campus. The experiments were carried out online with the Visual Basic application we wrote (see detail in the Procedure section). The three mazes differed only by the representations used to indicate direction: *Arrow*, *Line*, and *Juxtaposition* (boxes encasing an arrow inside). In order to assure that every route was equally difficult or equally easy the number of steps and number of turn from the starting point to the destination should be equal. This was not possible. Nevertheless, the maze was designed such that every route consisted of 28 to 31 steps and 6 to 8 turns including one backward turn. All three mazes can be found in Figure 4.1 and Appendix B-1.

Procedure

The Visual Basic application first described the three directional representations by examples, followed by a practice test that mimicked the experiment proper using three smaller sample mazes differed by directional representation. The mazes were described as consisting of routes that take the travellers, whose names were listed on the left-hand side of the maze, to one of the destinations on the right-hand side of the maze. During the practice session participants were free to ask questions and could do the practice test repeatedly. The maze program allowed them to start the experiment proper whenever they were ready after they had done at least one practice test.

In the experiment proper each participant was asked to answer nine questions, three per representation. The routine was as follows: on the first screen the participant was shown, for a few seconds only, the travellers and the destinations. Then the screen was blanked for two seconds before the incomplete maze was displayed again, but this time with a question asking the participant to give the destination for a specified traveller. When the participant clicked the mouse on the specified traveller's name, the missing routes appeared in the maze and the clock started measuring the time taken to answer the question. The participant then followed a route leading from the traveller in question to a destination city that they then pointed at and clicked with the mouse. The clock then stopped and the time for that participant and the task was recorded. Before moving to the next question, the participant was asked to confirm the answer and was allowed to change it. The answer was then recorded.

The order in which each directional representation was presented to participants was randomised. However, all three trials for the same representation were completed before another set of trials for a different representation followed. For example, a participant would be given three questions for the *Line* maze followed by three questions for the *Juxtaposition* maze, and finally three questions for the *Arrow* maze. For another participant the order of mazes might be different.

4.3.5 Results: Maze Study 1

The overall mean-score was 7.06. The group means were 7.00 and 7.12, for the Westminster and the Goldsmiths groups, respectively. The t-test statistics on total response time and total score revealed no significant difference between the two groups, $t(33) = 1.27$, ns and $t(33) = 0.14$, ns, respectively. Despite the simplicity of the task required of the participants, some scored as low as 2 out of 9 marks and only 42.9% achieved the full mark. This indicates that some participants might not have tried their best or did not spend enough time in the practice session to understand the rules and notations used. Therefore, the following data analyses were based on data from 24 participants whose total scores were 7 or above, 12 in each group.

Data analyses were subsequently conducted and described based on response time and accuracy performance. Two t-tests for unrelated data of the two groups of participants are presented first, followed by ANOVA results. Their means and the statistics for ANOVA, Q-tests, and t-tests are tabulated in Tables 4.1 and 4.2, respectively. The response time and accuracy performance data are also plotted in Figure 4.2.

Table 4.1 Maze Study 1: Mean response time and scores achieved by each participant

Representation	N	Response time		Accuracy	
		Mean	SD	Mean	SD
Arrow	24	6.67	3.1	2.96	0.2
Line	24	9.63	4.7	2.79	0.5
Juxtaposition	24	10.08	4.8	2.79	0.4

Response Time Analysis

A 2x(3x3) mixed ANOVA was performed for response time. The within-subjects factors were representation (3 levels: *Arrow*, *Line*, *Juxtaposition*) and trial (3 levels: Trial 1, Trial 2, Trial 3). The between-subjects factor is group (2 levels: Westminster and Goldsmiths). The ANOVA revealed main effects of representation and of trial. There was no interaction between trial and representation. Nor was there a between-subjects effect.

Pairwise comparisons of average response time taken over the three trials were conducted for three pairs of representation. The Bonferroni *p* value of 0.02 was used and the t-tests statistics revealed a significant difference between *Arrow* and *Line* and between *Arrow* and *Juxtaposition*. No significant difference between *Line* and *Juxtaposition* was found.

Accuracy Analysis

A one-factor ANOVA was performed. The independent variable was representation. The dependent variable was the sum of the scores over the three trials. ANOVA revealed no significant main effect of representation.

Cochran's Q-test for dichotomous nominal data analysis for score obtained in each trial was carried out for each representation separately. The Q-statistics revealed that accuracy performance was not significantly affected by trial number in any of the three representations.

Table 4.2 Maze Study 1: Directional representation statistics for response time and accuracy analyses

Factor	Response time analysis	Accuracy analysis
Representation	Within-subjects effect: $F\,(1.48, 32.55) = 8.18, p < 0.01$	Within-subjects effect: $F\,(2, 46) = 1.35$, ns
Trial	Within-subjects effect: $F\,(1.35, 29.65) = 5.52$, p < 0.02 Interaction: $F\,(1.97, 43.38) = 1.38$, ns	*Arrow*: Cochran Q = 2.0, df = 2, ns *Line*: Cochran Q = 3.5, df = 2, ns Jux: Cochran Q = 5.2, df = 2, ns
Group	Between-subjects effect: $F\,(1,22) = 0.42$, ns	
Significant difference		
Representation	*Arrow* - *Line*: $t(23) = -3.00$, p = 0.006 *Arrow* - Jux: $t(23) = -5.80$, p = 0.0005 *Line* - Jux: $t(23) = -0.43$, ns	

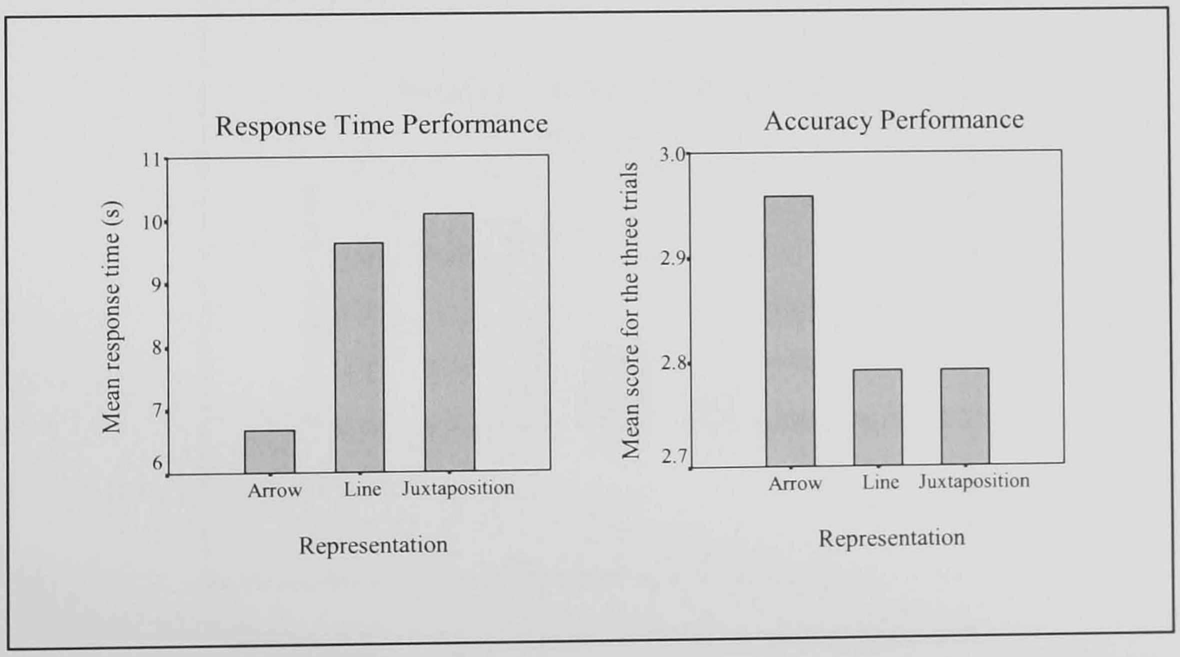


Figure 4.2 Maze Study 1: Response time and accuracy performance versus representation

4.3.6 Discussion: Maze Study 1

Hypothesis 1 (that response time and accuracy performance would be affected by representation) was supported. The ANOVA results found a main effect of representation in both response time and accuracy performance.

Hypothesis 2 (that *Arrow* would be the best performer) was not supported. *Arrow* outperformed both the *Line* and the *Juxtaposition* only in terms of response time. However, it was statistically inconclusive whether or not the accuracy performance of *Arrow* was better than that of *Line* and *Juxtaposition*. Nevertheless, the mean scores for *Arrow* was the highest suggesting that *Arrow* might be the best performer because it also gave the shortest response time.

This experiment had one flaw in it, however. A practice effect was found. Response time performance reduced with trial: the more practice, the shorter was the response time. Figure 4.3 illustrates this effect. This effect was due to the procedure of the experiment whereby participants were presented with the same representation three times in a row. Therefore, this experiment was repeated in Maze Study 2 with some modification to get rid of the practice effect.

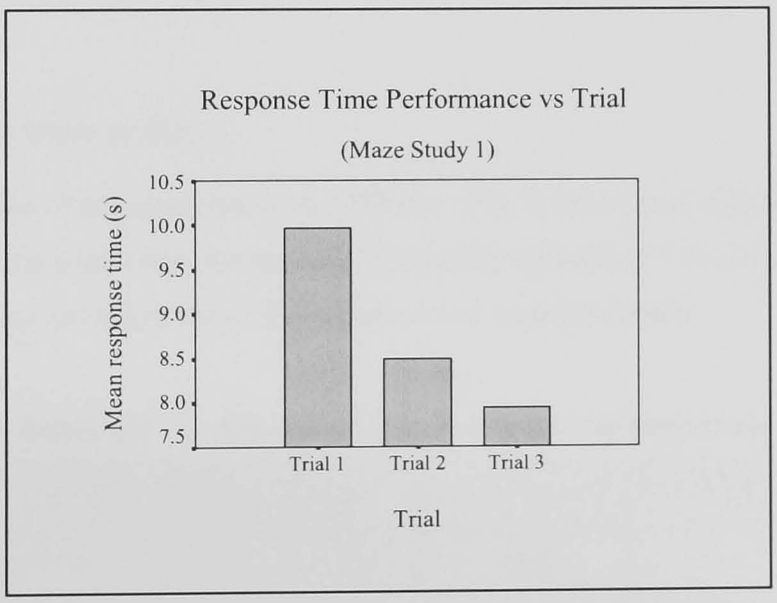


Figure 4.3 Maze Study 1: Practice effect observed

4.3.7 Method: Maze Study 2

Design and Materials

The same experimental design and materials as in Maze Study 1 were employed.

Participants

The participants were first year undergraduate Computer Science students, twenty-two from the University of Westminster (a different group of students from that which participated in Maze Study 1) and twenty-six from Brunel University. The Westminster group was learning Visual Basic whereas the Brunel group was learning the JAVA programming language.

Procedure

In this study, the procedure in Maze Study 1 was modified as follows. Instead of lumping all three trials for the same representation together, in the this experiment the order of the representation was completely randomised over all nine trials. Furthermore, participants were asked to fill in a questionnaire at the end. The post-hoc questionnaire asked them which one of the three representations they thought was the easiest and which the hardest.

4.3.8 Results: Maze Study 2

The outcome of the questionnaires ("Which of the three representations do you think was the easiest and which was the hardest?") based on 48 replies is shown in Table 4.3 below. The subjective rating showed that *Arrow* was most preferred.

Table 4.3 Questionnaire summary – in percentages

Opinion	Arrow	Line	Juxtaposition	No reply
Easiest	60.4	20.8	18.8	0
Hardest	10.4	47.9	37.5	4.2

The t-tests of the two groups on total response time and total score revealed no significant difference between the two groups, $t(46) = 0.213$, ns and $t(32.498) = 1.91$, ns, respectively. The means of total score were 6.55 for the Westminster group and 7.85 for the Brunel group. Total scores varied from 2 to 9 and 43.8% of all participants received the maximum score (9). The overall mean score was 7.25 and hence the following analyses use data obtained from 36 participants who scored 7 or above.

Response time and accuracy analyses subsequently described followed the analyses in Maze Study 1. Their means and the statistics for the ANOVA, Q-tests, and t-tests are tabulated in Tables 4.4 and 4.5, respectively. The response time and accuracy performance data are also plotted in Figure 4.4.

Table 4.4 Maze Study 2: Mean response time and scores achieved by each participant

Representation	N	Response time		Accuracy	
		Mean	SD	Mean	SD
Arrow	36	8.39	2.8	2.94	0.2
Line	36	11.61	6.2	2.94	0.2
Juxtaposition	36	13.09	5.9	2.58	0.6

<u>Response Time Analysis</u>

A 2x(3x3) mixed ANOVA was performed for response time. It revealed a main effect of representation but not of trial.

Pairwise comparisons of average response time taken for different types of representation were then made. The Bonferroni *p* value of 0.02 was used and the t-test statistics revealed a significant difference between *Arrow* and *Line* and between *Arrow* and *Juxtaposition*.

<u>Accuracy Analysis</u>

A one-factor, repeated-measures ANOVA was performed for the sum of the score over three trials on each representation. The independent variable was representation. A main effect of representation was found.

Cochran's Q-test for dichotomous nominal data analysis for score obtained in each trial was carried out for each representation separately. There was no main effect of trial in any of the three representations.

Pairwise comparison of scores obtained over the three trials for each type of representation was then made. The t-tests revealed significant differences both between *Arrow* and *Juxtaposition* and between *Line* and *Juxtaposition*.

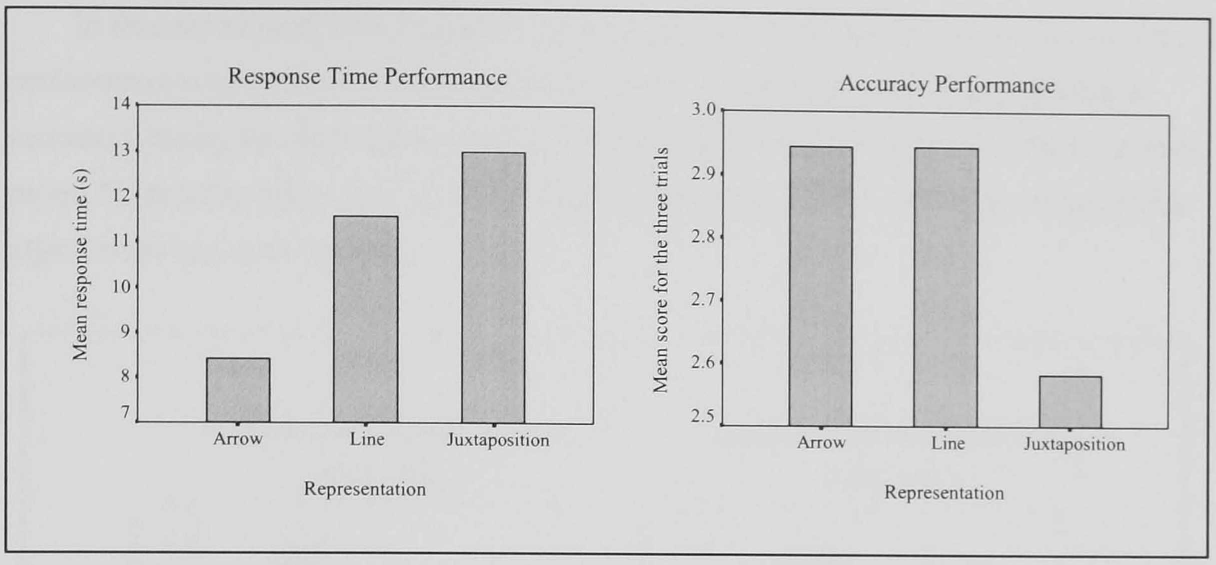


Figure 4.4 **Maze Study 2: Response time and accuracy performance versus representation**

Table 4.5 **Maze Study 2: Directional representation statistics for response time and accuracy analyses**

Factor	Response time	Accuracy
Representation	Within-subjects effect: $F\,(2, 68) = 15.97, p < 0.001$	Within-subjects effect: $F(1.43, 50.02) = 10.08$, p < 0.001
Trial	Within-subjects effect: $F\,(2,68) = 2.15$, ns Interaction: $F\,(2.35,79.73) = 1.84$, ns	*Arrow*: Cochran Q = 1.0, df = 2, ns *Line*: Cochran Q = 1.0, df = 2, ns *Jux*: Cochran Q = 0.46,df = 2, ns
Group	Between-subjects effect: $F\,(1,34) = 0.07$, ns	
Significant difference		
Representation	Arrow - Line: $t(35) = -3.70$, p = 0.001 Arrow - Jux: $t(35) = -5.80$, p = 0.0005 Line - Jux: $t(35) = -2.12$, ns	*Line - Jux*: $t(35) = 3.39$, p = 0.002 *Arrow - Jux*: $t(35) = 3.39$, p = 0.002

4.3.9 Discussion: Maze Study 2

In this experiment, both hypotheses were supported. Both response time and accuracy performance were affected by representation. *Arrow* was the best performer (speed and accuracy) among the three representations and *Juxtaposition* was found to be the most error prone. No practice effect was observed this time (see Figure 4.5 for the comparison of this effect in the two maze studies).

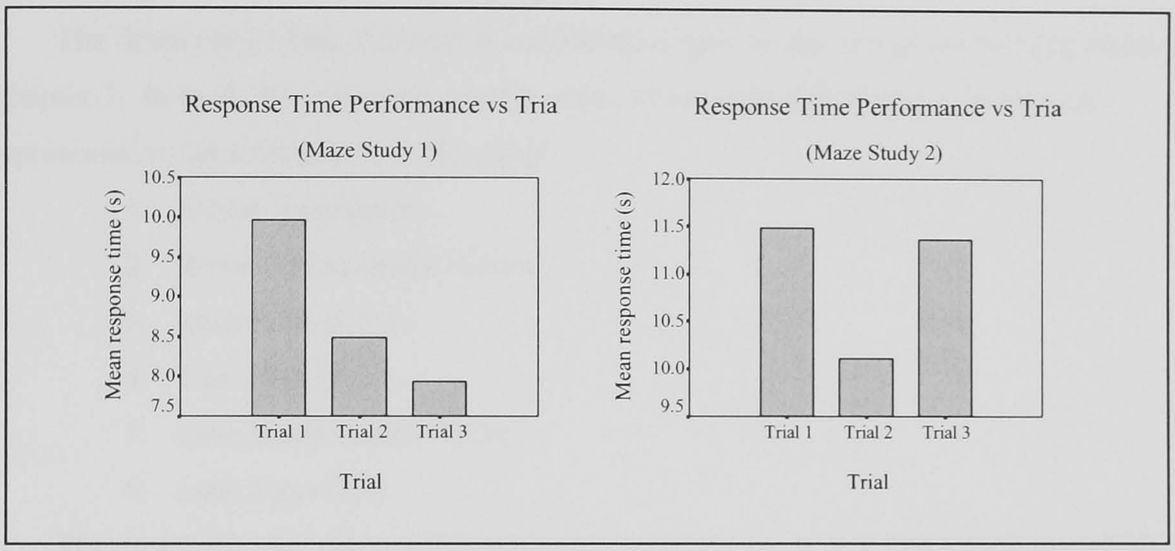


Figure 4.5 Practice effect observed in Maze Study 1 but not in Maze Study 2

4.3.10 General discussion

Response time performance in Maze Study 1 was consistently lower than that in the second experiment across all representations due to the practice effect found in Maze Study 1. No practice effect was observed in Maze Study 2. Despite this difference, the results from both studies were consistent. The two studies suggest that the choice of directional representation affects response time performance. *Arrow* outperformed both *Line* and *Juxtaposition* in response time performance in both studies. However, in terms of accuracy performance, the superiority of *Arrow* was evident only in the second study, in which *Arrow* was found to be the best performer and *Juxtaposition* the most error-prone. The results of the second study also agreed well with participants' subjective rating that *Arrow* was the easiest.

4.4 Flow Study 1

4.4.1 Objective

The purpose of this experiment is to compare the effects of directional representations and of traversal directions on comprehension performance in control flow visual programs.

4.4.2 General description

This experiment compared the performance of six visual programs, which differed by the combination of three traversal directions and two directional representations. The three traversal directions were Top-Down, Hierarchical-Nested, and Free-Style. The two directional representations were *Arrow* and *Line*. *Juxtaposition* was not included in this study because it was the most error-prone in the two maze studies presented in the previous section.

The description of the three traversal directions used in this experiment can be found in Chapter 3. In total, there were six combinations of traversal direction and directional representation for comparison in this study:

1. *Arrow*, Top-Down
2. *Arrow*, Hierarchical-Nested
3. *Arrow*, Free-Style
4. *Line*, Top-Down
5. *Line*, Hierarchical-Nested
6. *Line*, Free-Style

The programs were traversed by following lines or arrows. For Free-Style, graphical primitives were placed as randomly as possible within an acceptable degree of layout organisation. However, because the traversing was not restricted to be in any particular direction as in the case of Top-Down, participants could become confused when a line is used with the Free-Style. This could confound the experiment, as we only wanted to compare the effect of traversal direction. To solve the problem, a black spot was marked at one end of the line indicating where the line comes from. Examples can be found in the Appendices A-5.

4.4.3 Hypotheses

Six hypotheses were formed. Hypotheses 1 to 3 concerned the effect of traversal direction. Hypotheses 4 to 6 concerned the effect of directional representation.

Hypothesis 1. Based on the discussion in Chapter 2, we speculated that traversal direction would have an effect on both response time and accuracy performance.

Hypothesis 2. Hierarchical-Nested requires '*fall back*', which would increase the mental load for the participants and might cause them to forget to return to where they left off or to confuse them. We expected that this representation would perform less well than Top-Down in both response time and accuracy.

Hypothesis 3. Top-Down is similar to family trees or charts that are generally used in real life. As discussed in Chapter 3, participants should be more familiar with this style than other styles. Layout organisation in the Top-Down program was also good. Therefore, we expected this style to produce the best performances in both response time and accuracy.

Hypothesis 4. For Top-Down, we expected that directional representation would not matter too much due to familiarity, as discussed in Chapter 3.

Hypothesis 5. For Hierarchical-Nested, the use of *Arrow* seems not to be appropriate because it could cause one to forget to '*fall back*'. Line is bi-directional and hence we expected that *Line* would do better than *Arrow*.

Hypothesis 6. For Free-Style, we expected *Arrow* to do better than *Line* for two reasons. Firstly, the random traversal order and placement of the graphical primitives could benefit from the use of *Arrow* as a direction indicator. Secondly, with Line, participants may find it hard to remember the novel convention (reading from the black spot) used in this experiment for the Free-Style.

4.4.4 Method

Design

The experiment was a within-subjects design. Participants were subjected to all six visual program representations. The tasks required of them were answering forward and backward questions.

Participants

The participants in this experiment are the same twenty-two Brunel students who participated in the experiment presented in Chapter 3.

Materials

The experiment was administered online, using a 17-inch monitor, 1024 x 768 pixels resolution. The application that took participants through the whole experiment was written in Visual Basic by the author of this thesis. The detail of the experiment can be found in the Procedure section below.

Control flow programs were used in this experiment because we found better performance and lower variance with the control flow than with the data flow representation in the experiment presented in Chapter 3. The full textual program listing, its corresponding visual programs, and examples of forward and backward questions are given in Appendix B-2. Participants only saw its visual programs in three styles and two directional representations. A part of the textual version of the program used is given in Figure 4.6. The * symbol represents a wild card character. The program content was designed to be meaningless so that participants would not remember the answers or give answers based on their experience.

```
If S = '*Lettuce*' then
    Loop begins for Times = 1 to 2
        If S = '*Corn*' then
            Print 'Jupiter'
        End If
    End Loop
Else
    Print 'Uranus'
End If
```

Figure 4.6 Flow Study 1: A part of the textual program

Procedure

The procedure to train participants was given in Chapter 3. All participants were presented with all six visual programs and answered a total of 12 questions: six forward questions and six backward questions, one pair of forward and backward questions per program. The order of programs, questions, and question types seen by each participant was randomised. Participants never saw the same program or same question type on two consecutive trials. Participants could have a break between two consecutive trials. The first program and first question type of the series that each participant saw were also randomised. The application took the participants through the whole experimental procedure in a similar fashion to the one described in Chapter 3. Before the program ended, the program informed the participant of the total marks he/she achieved.

4.4.5 Results

Total scores ranged from 6 to 12 (the maximum). The mean score achieved was 10.45. The means for response time and scores per participant for the three traversal directions are given in Table 4.6. Table 4.7 tabulates statistics for ANOVA, t-tests, Q-tests, and McNemar tests performed in the subsequent response time and accuracy analyses. Response time and accuracy performance data are plotted in Figure 4.7. Table 4.8 provides a summary of the findings in this experiment.

Table 4.6 Mean response time (RT) and scores achieved by each participant

Traversal direction	N	Arrow		Line		Arrow		Line	
		RT	SD	RT	SD	Score	SD	Score	SD
Top-Down	22	58.64	26.4	75.95	46.4	1.82	0.39	1.86	0.35
Hierarchical-Nested	22	74.68	45.7	80.45	44.0	1.73	0.46	1.55	0.51
Free-Style	22	78.41	40.0	91.05	56.4	1.68	0.48	1.82	0.39

<u>Response Time Analysis</u>

Response time data were skewed; therefore a natural log function was applied to response time. There was a normal distribution in the transformed data. A two-factor, repeated measures ANOVA was performed on the transformed data. The two factors were traversal direction (three levels: Top-Down, Hierarchical-Nested, and Free-Style) and directional representation (two levels: *Arrow* and *Line*). The dependent variable was the sum of response times taken for both forward and backward questions. The ANOVA results revealed main effects of traversal direction and of directional representation. There was no interaction between traversal direction and directional representation.

Planned comparisons were performed between the following pairs:

- Top-Down, *Arrow* – Top-Down, *Line* (Hypothesis 4)
- Hierarchical-Nested, *Arrow* – Hierarchical-Nested, *Line* (Hypothesis 5)
- Free-Style, *Arrow* – Free-Style, *Line* (Hypothesis 6)
- Top-Down, *Arrow* – Hierarchical-Nested, *Arrow* (Hypothesis 2)
- Top-Down, *Arrow* – Free-Style, *Arrow* (Hypothesis 3)
- Top-Down, *Line* – Hierarchical-Nested, *Line* (Hypothesis 3)
- Top-Down, *Line* – Free-Style, *Line* (Hypothesis 3)

Unplanned comparisons were performed between the following pairs:

- Hierarchical-Nested, *Arrow* – Free-Style, *Arrow*
- Hierarchical-Nested, *Line* – Free-Style, *Line*

The t-tests indicated that Top-Down was superior to both Hierarchical-Nested and Free-Style for only the *Arrow*; and *Arrow* performed better than *Line* only in Top-Down. (The Bonferroni p value for the unplanned comparisons used was 0.03.)

Accuracy Analysis

The summed score of forward and backward questions was taken as the dependent variable. However these data were skewed. No transformation was possible to achieve non-skewed data as in response time data. Non-parametric tests were thus employed.

To test for the effect of traversal direction, since the data were dichotomous, Cochran's Q-test analysis was carried out for *Arrow* and *Line* separately. The statistics revealed no significant effect of traversal direction with *Arrow*. Hypothesis 3 was thus not supported for *Arrow*. However, there was a significant effect of traversal direction with *Line*. McNemar tests were then made for the following pairs:

- Top-Down, *Line* – Hierarchical-Nested, *Line* (Hypothesis 2, 3)
- Top-Down, *Line* – Free-Style, *Line* (Hypothesis 3)

Top-Down was found to significantly outperformed Hierarchical-Nested for *Line*. There was no significant difference between Top-Down and Free-Style for *Line*. Therefore McNemar test was not carried out for the 'Hierarchical-Nested, *Line* – Free-Style, *Line*' pair.

To test the effect of directional representation, McNemar tests for the following planned comparisons were carried out:

- Top-Down, *Arrow* – Top-Down, *Line* (Hypothesis 4)
- Hierarchical-Nested, *Arrow* – Hierarchical-Nested, *Line* (Hypothesis 5)
- Free-Style, *Arrow* – Free-Style, *Line* (Hypothesis 6)

The results did not reveal a significant difference in any of the three pairs for Top-Down, Hierarchical-Nest, and Free-Style.

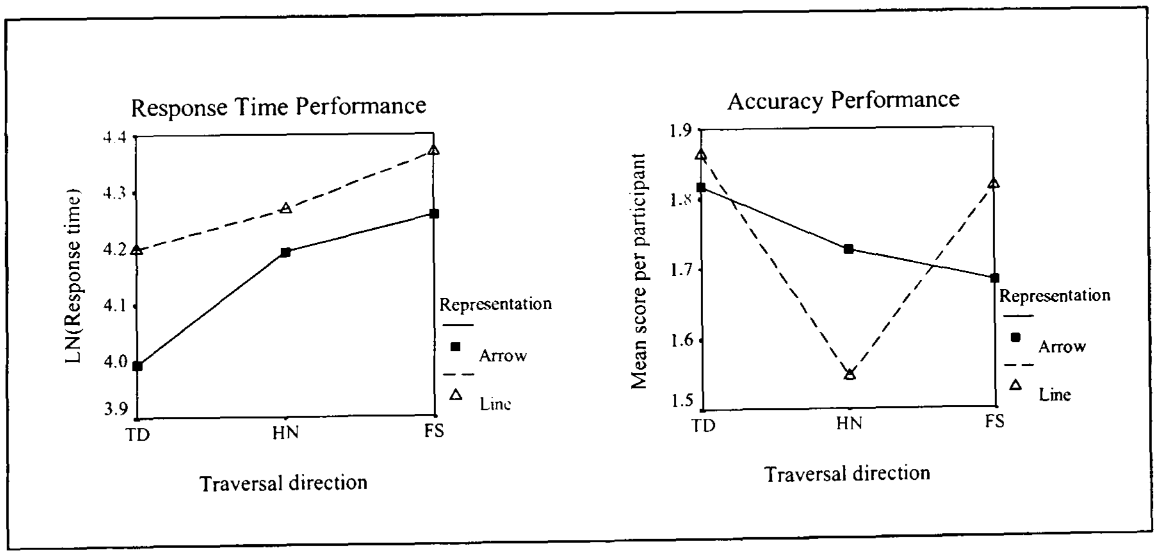


Figure 4.7 Flow Study 1: Effect of traversal direction and directional representation on response time and accuracy

(LN = Natural Logarithmic function; TD = Top-Down; HN = Hierarchical-Nested; FS = Free-Style; note that line graphs are used for readability)

Table 4.7 Flow Study 1: Flow representation statistics for response time and accuracy analyses

Factor	Response time		Accuracy	
Traversal direction	Within-subjects effect: F(2, 42) = 8.48, $p < 0.001$		Effect for *Arrow*: Cochran Q = 2.0; df = 2; ns Effect for *Line*: Cochran Q = 8.60, df = 22; p = 0.02	
Directional representation	Within-subjects effect: F(1,21) = 12.97, $p < 0.002$		Effect for TD: McNemar's p = 1, ns Effect for HN: McNemar's p = 0.22, ns Effect for FS: McNemar's p = 0.38, ns	
	Interaction, Traversal direction x Rep: F(2, 42) = 0.55, ns			
Significant difference				
Traversal direction:	TD -HN (Arrow)	*t*(21) = 3.34, p = 0.003		
	TD - FS (Arrow)	*t*(21) = 4.49, p = 0.0005		
	HN - FS (Arrow)	*t*(21) = 0.81, ns		
	TD - HN (*Line)*	*t*(21) = 0.80, ns	TD- HN (Line)	McNemar's p = 0.016
	TD-FS (*Line)*	*t*(21) = 1.97, ns	TD- FS (Line)	McNemar's p = 1.0, ns
	HN - FS (Line)	*t*(21) = 0.90, ns		
Representation	TD	*t*(21) = 3.14, p = 0.005	TD	McNemar's p = 1.0, ns
	HN	*t*(21) = 0.88, p = 0.39, ns	HN	McNemar's p = 0.22, ns
	FS	*t*(21) = 1.21, ns	FS	McNemar's p = 0.38, ns

(TD = Top-Down; HN = Hierarchical-Nested; FS = Free-Style)

Table 4.8 Flow Study 1: Summary of findings

Factor	Finding	Response Time	Accuracy
Traversal direction	Main effect	Yes	Yes (for *Line* only)
	Significant difference	TD better than HN for *Arrow* TD better than FS for *Arrow*	TD better than HN for *Line*
Directional representation	Main effect	Yes	
	Significant difference	*Arrow* better than *Line* for TD only	No

(TD = Top-Down; HN = Hierarchical-Nested)

4.4.6 Discussion

Hypothesis 1 (that traversal direction would affect both response time and accuracy) was supported. The ANOVA and Cochran's Q tests showed main effects of traversal direction in both response time and accuracy performance as expected.

Hypothesis 2 (that Hierarchical-Nested would perform poorer than Top-Down) was supported. Top-Down outperformed Hierarchical-Nested in both response time and accuracy performance.

Hypothesis 3 (that Top-Down would be the best performer) was partially supported. Top-Down was the best performer in both response time. This agrees with Curtis *et al.* (1989), who reported that forward response time performance of a branching spatial arrangement (which was similar to the Top-Down) performed significantly better than that of a hierarchical spatial arrangement (which had a '*fall back*' feature) in their study.

Hypothesis 4 (that directional representation would not have any effect on performance for Top-Down) was not supported. Directional representation mattered in Top-Down: *Arrow* outperformed *Line* in response time performance.

Hypothesis 5 (that *Line* would outperform *Arrow* in Hierarchical-Nested) and *Hypothesis 6* (that *Arrow* would outperform *Line* in Free-Style) were not supported. Directional representation was not found to have any effect on either Hierarchical-Nested or Free-Style.

Effect of directional representation

The finding regarding directional representation was inconclusive. *Arrow* was found to aid tracing speed but not accuracy. Figure 4.7 shows that *Arrow* took shorter time than *Line* across all representations but its accuracy performance was not consistent across these representations as in response time performance. This could be due to a design flaw in the *Line* representation used for Free-Style or to the inadequate length of the program.

Firstly, the *Line* used in Free-Style had a spot to indicate from where it originated. This, in effect, acted as a different type of arrow. The *Line* comparison made with Free-Style was then invalid as the same representation of *Line* had not been used. It is then necessary to repeat the experiment using the same *Line* representation across all traversal directions.

Secondly, Hierarchical-Nested used the same *Line* representation as Top-Down. So its behavioural pattern should be similar to Top-Down. On the contrary, while *Line* and *Arrow* accuracy performance did not differ much with Top-Down, there was a sharp drop in accuracy performance for *Line* from that for *Arrow* in Hierarchical-Nested. Did *Arrow* really aid tracing accuracy in Hierarchical-Nested, but not in any other? It may be because the program was deeply nested, but not long enough. Each branch (sub-hierarchy) had only one sub-branch. When '*fall back*' resumed, there was no descendant node or another sub-branch to trace. Had the program been longer (regardless of its depth), at a small circle in the diagram there would be two arrows: one pointing to the right and one downward, causing more confusion and hence lowering arrow performance. A lower effect of directional representation for Hierarchical-Nested than what was observed would probably result. The question still remains as to whether directional representation matters.

Differential carryover effect

In spite of its statistical power arising from controlling participants' variability due to individual differences, within-subjects design has some disadvantages. Two major ones are practice effect and differential carryover effect. This experiment was designed to eliminate practice effect by randomising the order that the diagrams were seen by participants. However, differential carryover effect cannot be controlled by counterbalancing. Differential carryover effect occurs when a preceding treatment condition affects a subsequent treatment condition in a different manner from how it would affect another subsequent condition (Keppel, 1991). We investigated the data and found that the accuracy performance of the (*Line*, Hierarchical-Nested) program dropped sharply when it was preceded by (*Arrow*, Top-Down) program and that the accuracy performance of the (*Arrow*, Hierarchical-Nested) program also dropped sharply when preceded by the (*Arrow*, Free-Style) program. Such was not the case with the other two traversal directions. Neither the performance of Top-Down, nor that of Free-Style was sharply affected by a preceding condition. Therefore, differential carryover effect did exist in this experiment due to the within-subjects design employed.

The '*fall back*' problem

Figure 4.7 shows a drastic drop in accuracy performance for *Line*. This could be due to the differential carryover effect discussed above. Or it was merely because *Arrow* performance was helped by the program being short. However, when considering this phenomenon with the finding that Top-Down outperformed Hierarchical-Nested in both response time and accuracy, it seems to suggest that Green's speculation (Green, 1982) about the '*fall back*' problem could be right. This is in contrast to our conclusion from the experimental results presented in Chapter 3 that Green's speculation was not supported. Therefore, his speculation needs more supporting evidence.

In sum, the findings indicate the effect of traversal direction on programmers' performance and possibility of the '*fall back*' problem. However, the results are not clear-cut due to some experimental design flaws. This suggests that the study ought to be repeated and the experiment be redesigned. Differential carryover effect that was present in this study suggests a between-subjects design for the future experiment. The inconclusive finding on the effect of directional representation that has been discussed suggests that it be further investigated and that the same line representation be used in all visual programs.

4.5 Flow Study 2

4.5.1 Objective

The main objective of this experiment is to improve the design of Flow Study 1 in order to achieve the goals listed below:

- To investigate and confirm the effect of '*fall back*' as indicated in Flow Study 1 (Section 4.4).
- To confirm the findings on directional representation in Maze Study 1 and Maze Study 2 (Section 4.3).
- To investigate the effect of traversal direction using layouts different from those used in Flow Study 1 (Section 4.4) and in the experiments presented in Chapter 3.

4.5.2 General description

This experiment compared the performance of ten visual programs, which differed by the combination of five traversal directions and two directional representations. The traversal directions were Top-Down, Hierarchical-Nested, Bowles, Rectangular-Net, and Curvy-Net. The two directional representations were *Arrow* and *Line*. The schematic diagrams of these traversal directions are shown in Figure 4.8.

The description of Top-Down, Hierarchical-Nested can be found in Chapter 3 and that of Bowles, Rectangular-Net, and Curvy-Net is described below.

<u>Bowles</u>

The program is traversed from the topmost and the leftmost primitive, all sub-levels of that primitive, then the next primitive to its right and its sub-levels, and so on until the rightmost primitive and its sublevels are traversed. Like Hierarchical-Nested, '*fall back*' is present.

<u>Rectangular-Net</u>

This is a form of Free-Style used before in the experiment presented in Chapter 3 and in Flow Study 1 (Section 4.4). The flow, represented by straight lines, is continuous but its direction is arbitrary. The overall shape is rectangular.

<u>Curvy-Net</u>

As the Rectangular-Net, this is another form of Free-Style used in previous experiments. The flow, represented by curved lines, is continuous but its direction is arbitrary. There is a trend that program flows towards the right hand side of the diagram. Its overall shape resembles a Yourdon style data-flow diagram (Yourdon, 1989).

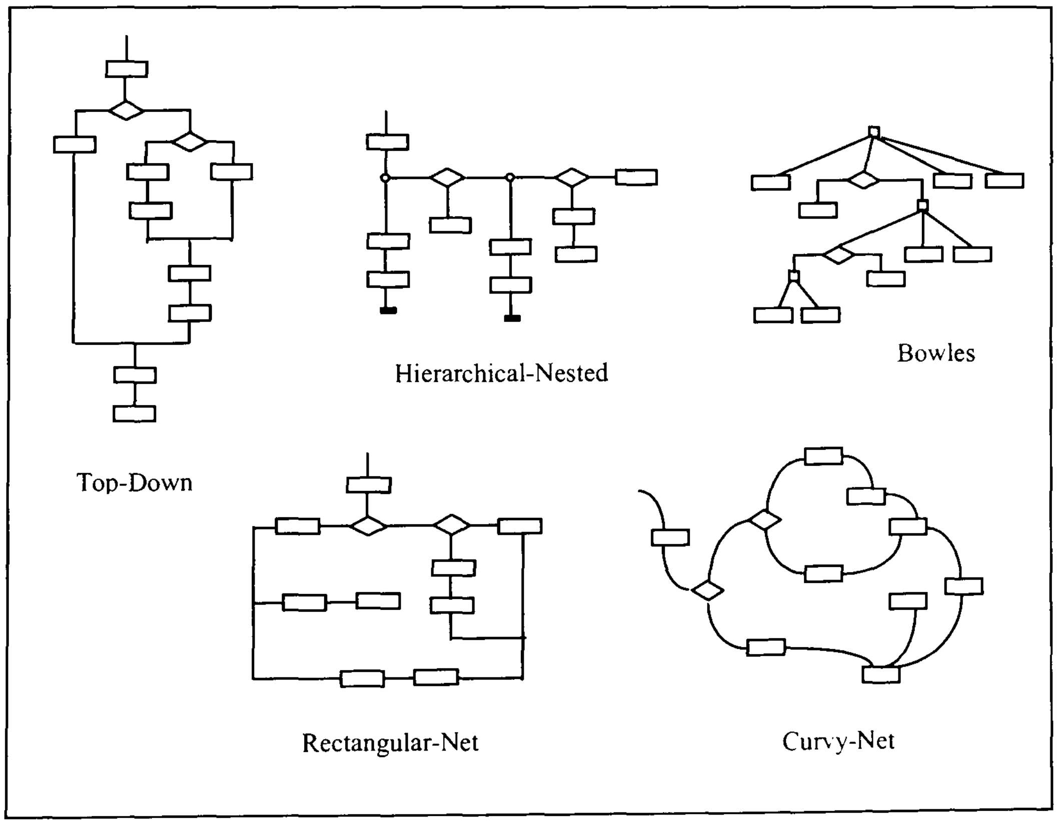


Figure 4.8 Traversal directions used in Flow Study 2

4.5.3 Sample size

The differential carryover effect found in Flow Study 1 (Section 4.4) suggests that a between-subjects design would be more appropriate. This means that participants are to be subjected to only one treatment condition (traversal direction). A mixed factorial design was then used in this experiment. The problem we faced was the fact that a between-subjects design is not as statistically powerful as a within-subjects design. Power analysis was therefore performed, based on the statistical data obtained from Flow Study 1 to estimate the sample size required for this experiment in order to achieve a reasonable power (0.80 or above). This could not have been done for the Flow Study 1 because there was no existing data available in the literature.

The sample size for this experiment was estimated from the effect size obtained from the F statistics for traversal direction effect on accuracy performance and the Pearson-Hartley Charts as recommended by Keppel (1991). The estimated sample size was 50 for an experiment with power = 0.8, and 60 at power = 0.9. However, this estimation was based on data for three traversal directions (three treatment conditions). Some projection still had to be done to get a better estimate for five traversal directions compared in this experiment. The estimation procedure (Keppel, 1991) involves the two statistics, ω^2 and Φ^2. The Φ^2 value varies with the value of ω^2. The value of ω^2 is an inverse function of the number of treatment conditions for large sample size. Increasing the number of treatment conditions reduces ω^2, and hence Φ^2. On the Pearson-Hartley Charts, to maintain the same power (0.8), the sample size (50) has to increase as Φ^2 decreases. Therefore the sample size required for this experiment must be larger than 50. We chose the estimated sample size of 60, which gave a power of 0.9 for three traversal directions to ensure a power of 0.8 in this experiment with five traversal directions, in case the power dropped due to the higher number of traversal directions compared here.

As for the effect of directional representation, the F-statistics from Flow Study 1 gave too small an effect size and too low a power to provide a good estimate of sample size. Furthermore, as the power was so low, the estimated sample size would have been too large to obtain. We therefore conducted this experiment using a sample size of 60, as estimated above. If this sample size could not reveal any effect from directional representation, the effect of directional representation could be interpreted as not being practically significant.

4.5.4 Hypotheses

As this is a repeated study for Flow Study 1, the hypotheses 1 to 6 in that study remain (see Section 4.4.3 for details). Here, however, two more hypotheses have been added, based on the findings of Flow Study 1.

Hypothesis 7. The results from Flow Study 1 indicated poor performance with the Hierarchical-Nested program. We expected that '*fall back*' would be a major factor that makes tracing difficult. Hierarchical-Nested and Bowles both have the '*fall back*' feature. Tracing thus requires that one place a 'mental finger' (Green, 1982) at the small circle to remind one where to '*fall back*' to. According to Green (1982), people seem to possess only one 'mental finger'. Once it is used for '*fall back*', one has no more 'mental fingers' left to be used for other tasks. Therefore, the mental load imposed upon him/her is high. We expected that Hierarchical-Nested and Bowles would be equally difficult and therefore their performance would not differ.

Hypothesis 8. We hypothesized that due to the '*fall back*' feature, Hierarchical-Nested would be outperformed by Top-Down, Rectangular-Net, and Curvy-Net.

4.5.5 Experimental design problems – practical issues

Given the difficulties in recruiting a large number of participants, we could only recruit first year undergraduate students in their first few weeks at Brunel University. We expected that the majority of the participants would have no programming experience and be unfamiliar with programming concepts and with reading diagrams typically used in the field of Computer Science. Therefore, training was necessary.

The experiment was administered with students in two JAVA laboratory sessions. Separate training sessions were not possible for financial reasons. Training had to be given in the laboratory just prior to the experiment proper. However, the experimental design required that participants be assigned to different traversal directions. It would not be possible to teach one group of participants in the laboratory what was assigned to them without the presence of the other groups. Subjecting others to knowledge of traversal directions other than what they were assigned to would confound the results. Therefore, we decided to include the appropriate training materials in the online application used for the experiment proper. Each participant would receive an online training relating only to the traversal direction that he/she was assigned to do in the experiment proper.

The success or failure of the experiment would depend very much on how well participants understood how to trace the programs and how to perform the tasks required without being personally tutored by the researcher. It was then only sensible not to introduce participants to too many programming concepts. The programs were then drawn based on the program used in the experiment presented in Chapter 3, which was a matching string problem. To make the program easier, the programs contained no loops. These programs were then tested with a volunteer who had no programming background at all. Another problem unfolded. The 'Matching a string' problem was incomprehensible to non-programmers. The pen-and-paper based pilot test took three hours and required constant communication between the volunteer and the researcher. We then decided to use familiar or story-based scenarios for the programs to be used for training and the experiment proper. For the practice test, we used a program scenario based on aliens travelling to Uranus. For the experiment proper, the program we used was based on a supermarket shopping scenario. Another round of pen-and-paper based pilot test was then run with three volunteers, all of whom had absolutely no knowledge of programming concepts or flowcharts. The question-answering tasks for both forward and backward questions and the set of questions used in this pilot test were the same as the ones used in the experiment proper. Only the Top-Down diagram was used. None of the volunteers had any problems understanding the training materials on their own. Their scores were 86%, 94%, and 98% correct. These two newly devised programs were then used for the training materials and the experiment proper.

4.5.6 Pre -Test

Because the design of the experiment was a mixed factorial and the between-subjects factor was traversal direction, it was important that participants' variability due to individual differences was reduced as much as possible. One week before the experiment, we gave 79 participants (all the students who attended the laboratory session at the time) a cognitive test as a pre-test so that we could use the results to assign participants into groups of about the same average cognitive ability. The Choosing a Path Test, taken from the Kit of Factor-Referenced Cognitive Tests (Ekstrom *et al.*, 1976) was used. Due to time constraint only the first of the two tests of the Choosing a Path Test was used. The test consisted of sixteen questions. Participants were given seven minutes to do the test as required by the Kit. Samples of the test can be found in the Appendix B-3. The results from this test were then used to randomly assign participants to each group to maintain equal average cognitive test performance.

4.5.7 Post-Hoc Questionnaire

A one-page questionnaire, which can be found in the Appendix B-3, was given to the participants in the week that followed the experiment proper. It was designed specifically for carrying out a discriminant analysis of participants' prior experiences. Forty-one questionnaires were returned and the results are presented in Section 4.5.9.

4.5.8 Method

Design

The experiment was a mixed factorial design. Participants were divided into five groups. Each group was subjected to only one traversal direction but to both directional representations. The between-subjects factor was traversal direction and the within-subjects factor was directional representation.

Participants

Sixty-three Brunel students from the Department of Information Systems and Computing participated in this experiment. The students were in their third week of their first year and were taking the introductory course in JAVA programming. They had just had two weeks of HTML and were in the first week of JAVA programming.

Out of the 79 participants in the two classes who took the cognitive test only 41 voluntarily participated in the experiment proper. Upon examining the data of these participants, we found no correlation between their cognitive test results and accuracy performance in the experiment proper. Due to the difficulty of recruiting students to do both tests voluntarily, the rest of the participants were then recruited from another JAVA programming class for the experiment proper only.

Materials

Programs

The program used consisted of only conditional structures and was based on the Supermarket Shopping scenario as discussed in Section 4.5.5. The textual program can be found in the Appendix B-3.

Questions

The task required of participants was question-answering for both forward and backward questions as in the experiment in Chapter 3 and in Flow Study 1. An example of the forward and backward questions can be found in Chapter 3 (Figure 3.3).

The application program

The experiment was administered online, using a 15-inch monitor, 1024 x 768 pixels screen resolution. The program was written in Visual Basic. It recorded the response time and answers from the participants.

Procedure

Participants were randomly assigned in advance by group (traversal direction), representation order (order in which *Arrow* or *Line* representation is seen first), and set (order of the questions seen). Half of the participants were presented with the *Arrow* program before the *Line* program, and the other half, with the *Line* program before the *Arrow* in order to counter-balance any order effect. The application walked them through a training session, which described the program and the symbols used and showed by example how to extract information from it. Then it gave an online practice session that mimicked the real test using the program that was used in Flow Study 1 and asking six questions, forward and backward. Both *Arrow* and *Line* programs were presented. At the end of the practice test, the program informed the participant his/her score.

In the experiment proper, every participant answered eight forward questions and eight backward questions for one traversal direction, half of the times in *Arrow* and the other half in *Line* representation, the order of which was assigned in advance. The order of questions, and question type seen by each participant was randomised and alternated, respectively. The application took the participants in a similar fashion as in Flow Study 1 (Section 4.4). Before the program ended, the program informed the participant the total marks he/she achieved.

4.5.9 Results

Cognitive test

Seventy-nine students took the cognitive test but only 41 participated in the experiment proper in the following week. Table 4.9 gives the correlation statistics of participants who both took the cognitive test and the experiment. Pearson correlation between the cognitive test scores and the experimental scores was insignificant in all groups.

Table 4.9 Mean scores of the cognitive test scores and the experimental scores

Group	N	Cognitive test (% correct)	Experiment proper (% correct)	Correlation between the two tests
TD	8	26	84	No
HN	7	31	78	No
BS	7	30	73	No
RN	8	42	87	No
CN	11	25	89	No
Overall	41	30	83	No

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

Post-Hoc Questionnaire

Since there was no correlation between the cognitive test and the experimental performance, a one-page questionnaire (Appendix B-3) was given out to the participants in the following week. Its purpose was to give us more information about participants' prior experience which might have affected their performance. Upon finding what prior experience(s) that could be, we would therefore be able to check whether that experience was approximately equal across all groups.

Programming experience

Of the sixty-three participants participating in the experiment proper, forty-one responded. The questionnaire results showed that participants had known an average of 0.64 programming languages. Forty-eight percent had no previous programming experience and 16% self reported as being good at programming.

Discriminant analysis

A Discriminant analysis was conducted from the questionnaire data. The independent variables taken from the questionnaire questions are:

1. Previous programming experience
2. Academic achievement
3. Interest in board games
4. Map reading skill
5. Experience with computer games and Nintendo games
6. Interest in D.I.Y.
7. Interest in Drawing
8. Interest in construction toys such as Lego
9. Having a PC at home or not
10. Gender
11. Previous experience with flow diagrams

The dependent variable was the experimental accuracy performance broken down into four levels according to which quartile the participant's performance belongs. The analysis gave only one function and one independent variable for the discriminant function with 48.8% success rate in classification. The Discriminant function was the 'Interest in construction toys' such as Lego. It turned out that the mean-score for 'Interest in construction toys' across groups did not differ much. On a scale 1 to 5, the group means ranged from 3.6 to 4.0.

Effect of traversal direction

Data from 60 participants were analysed after removing three outliers. There were 12 participants in each group. The overall mean-score achieved was 82%. Data analyses were made for response time and accuracy (% correct) separately. The means of response time and scores for each group can be found in Table 4.10. The data in this table indicate that TD is not the best performer in both response time and accuracy as we expected. Table 4.11 tabulates the ANOVA and the t-test statistics for both response time and accuracy performance. Figure 4.9 plots both the response time and accuracy performance.

Table 4.10 Descriptive Statistics for both accuracy and response time performances

Group	N	Response time per question (s)		Accuracy (% correct)	
		Mean	SD	Mean	SD
TD	12	63	18	87	9
HN	12	68	15	73	18
BS	12	70	18	74	12
RN	12	50	15	86	7
CN	12	57	10	89	8

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

Response Time Analysis

A 5x(2) mixed ANOVA was performed. The between-subjects factor was traversal direction (five levels: Top-Down, Hierarchical-Nested, Bowles, Rectangular-Net, and Curvy-Net) and the within-subjects factor was directional representation (two levels: *Arrow* and *Line*). The dependent variable was the average response times per question. The ANOVA results revealed a between-subjects effect of traversal direction. No main effect of directional representation was found. Nor was there an interaction between directional representation and traversal direction.

Planned comparisons were performed on overall response time taken by both Arrow and Line to test hypotheses 7 and 8. The t-test statistics revealed that Hierarchical-Nested was significantly slower than Rectangular-Net, and marginally slower than Curvy-net. However, it was not significantly slower than Top-Down or Bowles.

Unplanned comparisons were then made for Bowles, the Bonferroni *p* value used was 0.02. The t-test results revealed that Bowles was significantly slower than Rectangular-Net only.

Accuracy Analysis

The same data analysis procedure as in the response time analyses was performed. The dependent variable was % correct answers. The 5 x (2) ANOVA results revealed between-subjects effect of traversal direction. No main effect of directional representation was found. There was no interaction between directional representation and traversal direction.

Planned comparisons of the overall accuracy performance of both *Arrow* and *Line* revealed that the performance of Hierarchical-Nested was significantly poorer than that of Top-Down, Rectangular-Net, and Curvy-Net. There was no significant difference between Hierarchical-Nested and Bowles.

Unplanned comparisons, using the Bonferroni *p* value of 0.02, revealed that Bowles was significantly poorer than Top-Down, Rectangular-Net, and Curvy-Net.

Table 4.11 Flow Study 2: Flow representation statistics for response time and accuracy analyses

Factor	Response time	Accuracy
Traversal direction Directional representation	Between-subjects effect: F(1, 4) = 3.40, *p* < 0.02 Within-subjects effect: F(1,55) = 0.37, ns Interaction: F(4, 55) = 0.2, ns	Between-subjects effect: F(1, 4) = 5.46, *p* < 0.001 Within-subjects effect: F(1,55) = 2.41, ns Interaction: F(4, 55) = 1.18, ns
Significant difference		
t-tests	HN - BS: *t*(22) = -0.39, ns HN - TD: *t*(22) = 0.72, ns HN - RN: *t*(22) = 2.86, p = 0.009 HN - CN: *t*(22) = 2.05, p = 0.053 BS - TD: *t*(22) = 1.02, ns BS - RN: *t*(22) = 2.94, p = 0.008 BS - CN: *t*(22) = 2.20, p = 0.039, ns	HN - BS: *t*(19.40) = -0.18, ns HN - TD: *t*(16.02) = -2.54, p = 0.022 HN - RN: *t*(14.01) = -2.40, p = 0.031 HN - CN: *t*(15.18) = -2.85, p = 0.013 BS - TD: *t*(22) = -3.11, *p* = 0.005 BS - RN: *t*(17.07) = -3.02, p = 0.008 BS - CN: *t*(22) = -3.56, p = 0.002

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

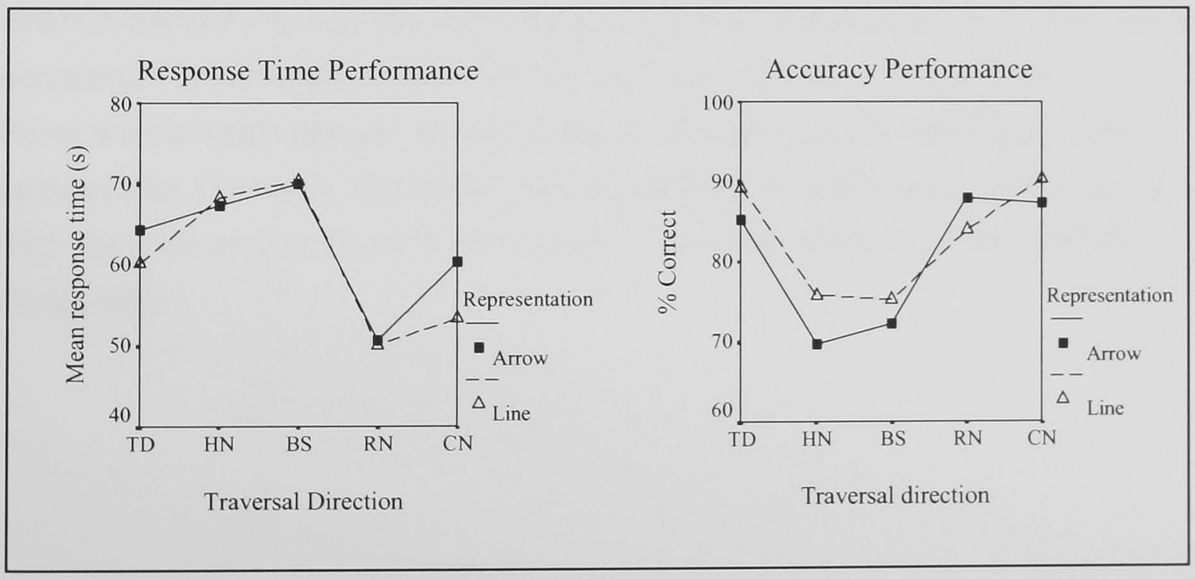


Figure 4.9 Accuracy and response time performance

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

Effect of question type

In chapter 3 the 'Match-Mismatch effect' was not observed for visual programs. Here we have another opportunity to confirm the finding. The analyses of question type effect are described below. The ANOVA, t-test statistics, and the summary of the findings can be found in Tables 4.12 and 4.13, respectively.

Response time analysis

Because there was no main effect of directional representation, a 5x(2) mixed ANOVA was conducted as follows. The within-subjects factor was question type and the between-subjects factor was traversal direction. The dependent variable was the sum of response time taken by *Arrow* and *Line*. The ANOVA revealed main effects of question type and traversal direction. However, there was an interaction between question type and traversal direction. Pairwise comparisons between forward and backward response time performance was conducted for all traversal directions. The Bonferroni p value for five tested pairs was 0.01. The t-tests showed that backward tracing took significantly longer than forward tracing in all the traversal directions.

Accuracy analysis

Following the procedure in the response time analysis, the percentage of the sum of scores for both *Arrow* and *Line* was used as the dependent variable. The 5x(2) mixed ANOVA revealed a strong main effect of question type and traversal direction. There was no interaction. The between-subjects effect was significant, $F(1,4) = 4.03, p < 0.006$. Pairwise comparisons between forward and backward response time performance were performed for Top-Down, Hierarchical-Nested, and Bowles. Bonferroni p value used was 0.02. The t-test statistics (Table 4.12) revealed a significant difference for Hierarchical-Nested only.

Table 4.12 Flow Study 2: Question type statistics for response time and accuracy analyses

Factor	Response time	Accuracy
Question type	Within-subjects effect: F(1, 55) = 101.62, p < 0.001	Within-subjects effect: F(1,55) = 13.63, *p* < 0.001
Traversal direction	Between-subjects effect: F(1, 4) = 2.76, *p* < 0.04	Between-subjects effect: F(1, 4) = 4.03, *p* < 0.006
	Interaction, Question type x Traversal direction: F(4,55) = 3.19, *p* < 0.02	Interaction, Question type x Traversal direction: F(4,55) = 1.57, ns
t-test comparisons between forward and backward question		
TD	*t*(11) = 5.21, *p* = 0.0005	*t*(11) = 2.19, *p* = 0.05, ns
HN	*t*(11) = 4.03, *p* = 0.002	*t*(11) = 3.71, *p* = 0.003
BS	*t*(11) = 4.89, *p* = 0.0005	*t*(11) = 2.38, *p* = 0.04, ns
RN	*t*(11) = 6.36, *p* = 0.0005	not performed
CN	*t*(11) = 5.04, *p* = 0.0005	not performed

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

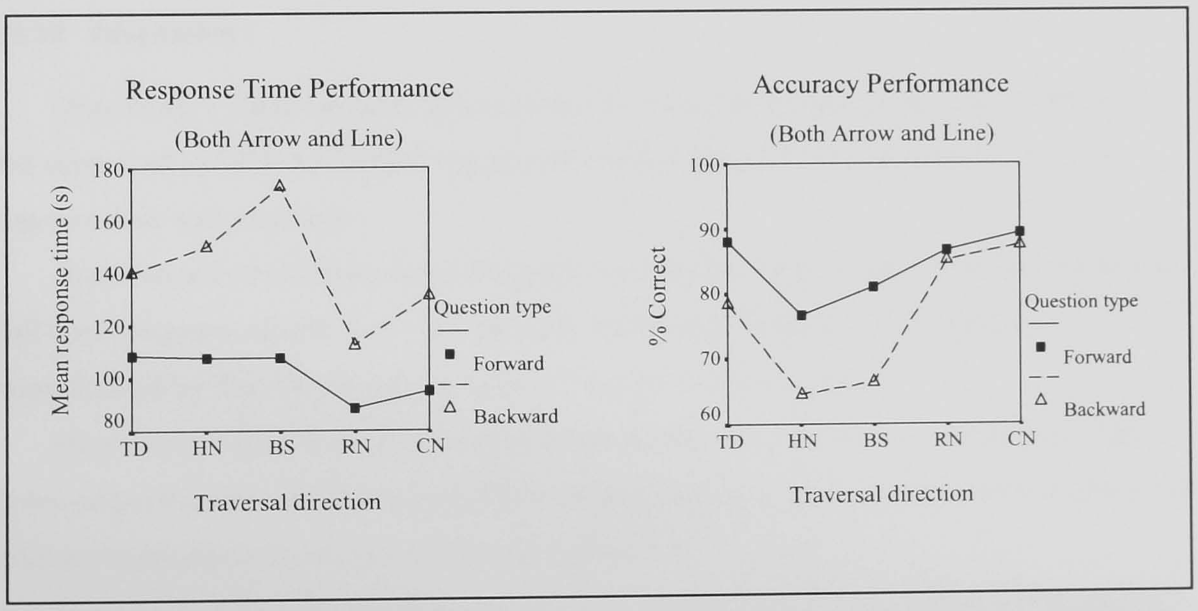


Figure 4.10 Forward and backward performance for *Arrow* diagrams

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net and note that line graphs are used for readability purpose only.)

Table 4.13 Flow Study 2: Summary of findings

Factor	Finding		Response time	Accuracy
Traversal direction	ANOVA main effect		yes	yes
	Significant difference:	HN - BS	no	no
		HN - TD	no	yes
		HN - RN	yes	yes
		HN - CN	marginal	yes
		BS - TD	no	yes
		BS - RN	yes	yes
		BS - CN	no	yes
Question type	ANOVA main effect		yes	yes
	Significant difference:	TD	yes	no
		HN	yes	yes
		BS	yes	no
		RN	yes	no
		CN	yes	no
Directional representation	ANOVA main effect		no	no

(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

4.5.10 Discussion

Hypothesis 1 (that traversal direction would affect both response time and accuracy) was supported. ANOVA revealed significant main effects of traversal direction in both response time and accuracy.

Hypothesis 2 (that Hierarchical-Nested would perform poorer than Top-Down because '*fall back*' requires mental load) was partially supported. Hierarchical-Nested was outperformed by Top-Down only in terms of accuracy performance.

Hypothesis 3 (that Top-Down would be the best performer) was not supported. Top-Down outperformed only Hierarchical-Nested and Bowles only in accuracy performance but it did not outperform Rectangular-Net and Curvy-Net.

Hypothesis 4 (that directional representation would not have any effect on performance for Top-Down) was supported. ANOVA main effect of directional representation was not found.

Hypothesis 5 (that *Line* would outperform *Arrow* in Hierarchical-Nested) and *Hypothesis 6* (that *Arrow* would outperform *Line* in Free-Style, or Rectangular Net and Curvy Net in this study) were not supported for the same reason as in Hypothesis 4.

Hypothesis 7 (that Hierarchical-Nested and Bowles are equally difficult and their performance do not differ) was supported. There was no significant difference between the performances of the two traversal directions.

Hypothesis 8 (that Hierarchical-Nested would be outperformed by Top-Down, Rectangular-Net, and Curvy-Net because of the '*fall back*' feature) was supported in terms of accuracy performance only. The accuracy performance of Hierarchical-Nested was significantly poorer than Top-Down, Rectangular-Net, and Curvy-Net.

Further discussion of the findings and the design of the experiment are presented below.

Experimental design assessment

The experiment presented in Chapter 3 was useful to the design and the success of this experiment. Its findings indicate a control flow preference among novice participants. Therefore the visual programs used in the present experiment were control flow based. It appeared that participants were able to cope with learning the experimental procedure, programming concepts, and how to trace the visual programs online and on their own. The mean score was as high as 82%. This outcome might have been different had a control flow program not been used.

Data from Flow Study 1 presented in Section 4.4.5 have been helpful for the design of this experiment. Power analysis proved to be useful in estimating the sample size for this experiment from the data obtained from that experiment. We estimated a sample size of 60 to ensure power of 0.8. The SPSS data show the power of this experiment to be 0.819. The experiment was powerful enough to reveal the effect of traversal direction.

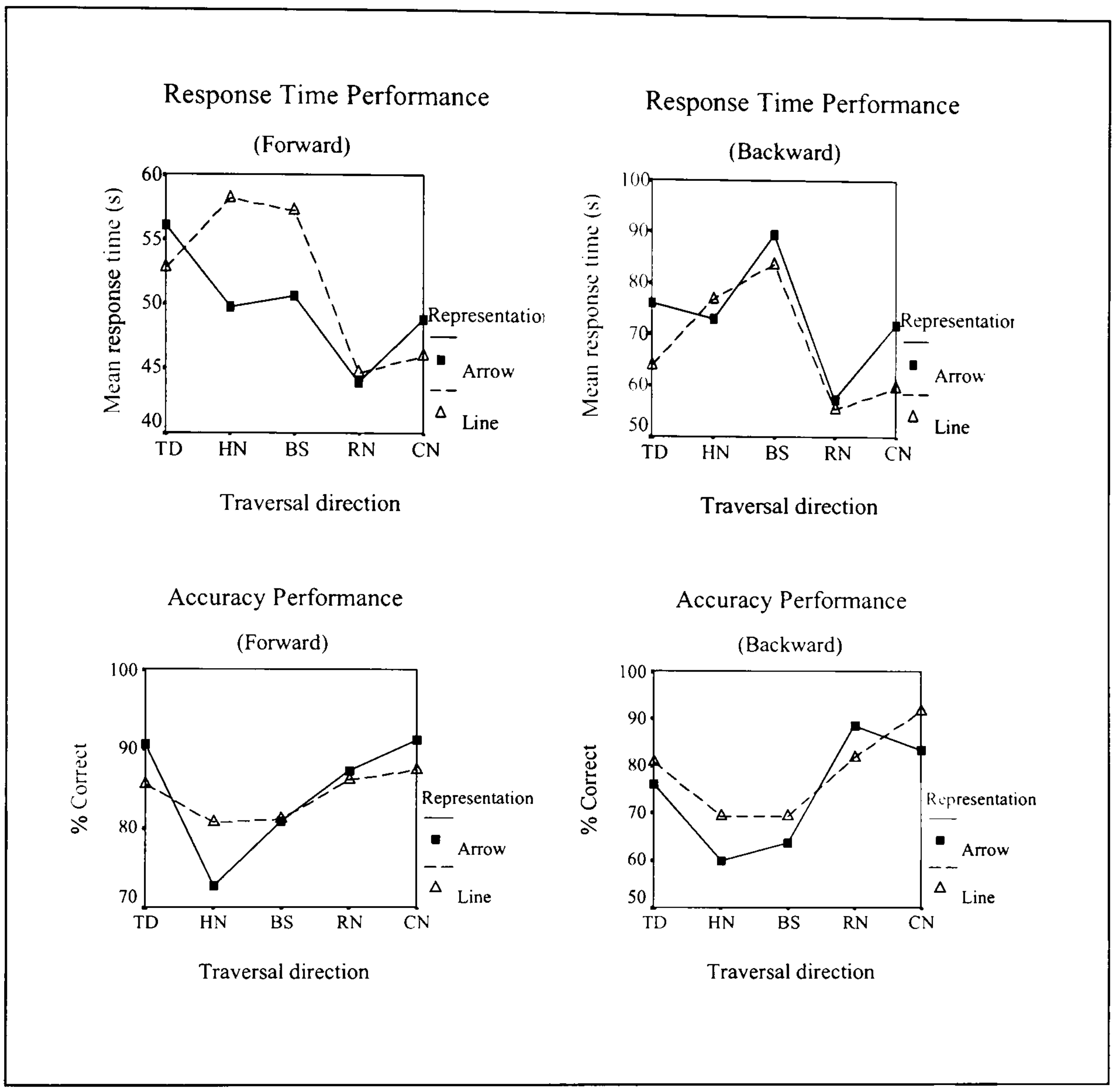
The speculation in Flow Study 1, made in Section 4.4.6, that the program was too short and hence gave the benefit to (*Arrow*, HN) was confirmed in this experiment. In this experiment (*Arrow*, HN) did not outperform (*Line*, HN) in terms of accuracy, contrary to the result in that study.

The differential carryover effect observed in the study in Section 4.4.6 did not materially confound its results. The same effect of '*fall back*' was still observed in this experiment.

The average scores of 'Interest in construction toys' were quite consistent across the experimental groups. Since it was the only variable in the Discriminant function out of 11 variables investigated, we argue that average ability for doing the experiment proper was also consistent across the groups. Hence, individual difference between groups had been minimised as much as possible.

Choice of directional representation

It was found that choice of directional representation did not affect overall performance in this experiment. On the contrary, results in the Maze experiment presented in Section 4.3 indicated that *Arrow* performed better than *Line*. The reason for this discrepancy may be that in the maze studies participants performed only the forward tracing task. Perhaps *Arrow* would be better than *Line* for forward tracing. *Arrow* is very good at pointing in the forward direction and hence, enhances forward tracing performance. At the same time, it could cause confusion in backward tracing task. Therefore had both forward and backward tracing tasks been performed in the Maze Studies, the advantage of *Arrow* over *Line* observed in the Maze experiment might have diminished. To confirm this, we conducted a 5x(2x2) mixed ANOVA, for both response time and accuracy performance. The two within-subjects factors were question type (2 levels: forward and backward) and directional representation (2 levels: *Arrow* and *Line*). The between-subjects factor was traversal direction. There was no main effect of directional representation found. *Arrow* and *Line* performed equally well in both forward and backward performance. The data that are plotted in Figure 4.11 show that *Arrow* seems to give a better response time performance in forward tracing only for Hierarchical-Nested and Bowles. Nevertheless, the t-tests for these two pairs did not reveal a significant difference. In short, *Arrow* was not better than *Line*, even in forward tracing. Perhaps, therefore, the Maze paradigm was not representative for studying the effect of graphical representation in a programming problem.



(TD = Top-Down; HN = Hierarchical-Nested; BS = Bowles; RN = Rectangular-Net; CN = Curvy-Net)

Figure 4.11 Forward and Backward performance of *Arrow* and *Line*

Effect of traversal direction

The effects of the traversal direction were observed for both the response time and the accuracy of performance. Hierarchical-Nested and Bowles were equally hard. In terms of accuracy, all other diagrams tested outperformed Hierarchical-Nested and Bowles. This confirms our prediction that traversal direction affects performance and that '*fall back*' is a crucial factor that affects the cognitive demand on the user. '*Fall back*' is therefore definitely an undesirable feature.

The 'Match-Mismatch' phenomenon

The results from Flow Study 2 give a strong evidence of the 'Match-Mismatch' phenomenon in visual programs. The 'Match-Mismatch' effect has been found in textual programs but not in visual programs, as has been discussed in Chapter 2 and again in our own results of the experiment presented in Chapter 3. What then could be an explanation for this discrepancy?

Good's (1999) explained that the Match-Mismatch effect was not observed in her first experiment because 'control flow supremacy' (that best performance was obtained for control-flow representation or tasks) overrode the Match-Mismatch effect. This could explain the contradictrary results for VPLs in the literature. The Match-Mismatch phenomenon may indeed be easily overridden by other factors.

Curtis *et al.* (1989) presented the diagrams on a sheet of paper to their participants. Therefore, participants could see all parts of the program they were working at simultaneously. Based on the paper by Moher *et al.* (1993), we inferred that the diagrams seen by their participants occupied one screen per program. The experiment in the Chapter 3 used a short program. Though it was deeply nested, scrolling was hardly required. The common factor among all these programs is visibility. Compared to them, the programs in this present experiment, where the 'Match-Mismatch' was found, had poor visibility. We therefore conclude that the reason why the 'Match-Mismatch' was not apparent in visual programs before was due to visibility overriding the 'Match-Mismatch' effect.

Graphical readership skills

All participants in this experiment were new entrants to Brunel University. They brought with them individual differences due to their prior experience. Although diagram reading skill depends on experience among experts (Petre & Green, 1993), it has not been known what previous experience, other than being familiar with the diagram convention, affects the ability to read diagrams among novices. Results from the Discriminant Analysis we conducted showed that previous experience in programming and flow diagrams, which seem to be the most likely candidates to affect the experimental results, were not the predicting variables for the diagram reading ability. Of all the previous experiences questioned in the post-hoc questionnaire, 'Interest in construction toys' was found to be the best candidate for predicting diagram reading ability. Lego toys are supplied with diagrammatic instructions of how to build objects such as cars, trucks, aeroplanes etc. To be good at playing construction toys, one must have a lot of experience in reading these diagrams. So this result provides a support to Green & Petre (1993) who state that diagram reading skill can be trained over time.

Control flow bias among novices

This study supports the finding in Chapter 3 that a control flow preference seems to exist among our participants. The visual programs that our novice participants were subjected to in the experiment were control flow based. Despite the fact that they were inexperienced with programming, had little time to train themselves in the experiment tasks and the programming related concepts, they coped with the experiment in Flow Study 2 quite well. The average of the mean scores was 82 %. This is a good indication that control flow programs are not difficult.

Effect of scrolling

The effect of scrolling is not to be neglected. It appeared to have some effect on response time performance for the diagrams without the '*fall back*' feature (Top Down, Rectangular-Net, and Curvy-Net). The Rectangular-Net participants appeared to have taken less time to complete the tasks than the Top Down and the Curvy-Net participants. In this study Rectangular-Net had only one primitive to be scrolled for while both Top-Down and Curvy-Net had six and seven, respectively. The scrolling effect was not observed across all five diagrams, however. It appeared that the effect of '*fall back*' was more dominant. The Hierarchical-Nested program had the same number of items to scroll for as the Rectangular-Net (1) and much lower numbers than Curvy-Net (7) and Top-Down (6). However, both its response time and accuracy performance were significantly worse than all three of them.

4.6 Chapter summary

This chapter presented four experiments that compared novices' response time and accuracy performance in tracing visual programs, forward and backward, using different representations for direction and for traversal direction. In the first two experiments (Maze Study 1 and Maze Study 2), *Arrow*, *Line*, and *Juxtaposition* were compared on the merits as directional indicators. *Arrow* was found to be the best indicator in the forward direction while *Juxtaposition*, the most error-prone. Based upon these two studies, only *Arrow* and *Line* were used and compared in subsequent studies: Flow Study 1 and Flow Study 2. These two studies compared the effects of both directional representation and traversal direction on novices' performance within the same programs. The effect of directional representation was found to be inconclusive in Flow Study 1 due to a design flaw with the *Line* representation used for one of the traversal directions, as discussed earlier. Although there was some indication that traversal direction and '*fall back*' affected performance, this was not clear either. This was possibly due to a differential carryover effect observed in the within-subjects experiment conducted. Therefore the experiment was redesigned and carried out in the second study: Flow Study 2. This latter study gave a clear-cut conclusion that *Arrow* and *Line* did not affect performance differently while traversal direction did and that the crucial factor was the '*fall back*' feature inherent in the traversal direction that imposed high cognitive demand on novices. Green's (1982) arm-chair analysis that '*fall back*' would make the programs so difficult to trace is thus confirmed.

Many other interesting issues have also been revealed (Chattratichart & Kuljis, 2001). Firstly, the 'Match-Mismatch' phenomenon was observed in Flow Study 2. This provides evidence supporting the applicability of the research in the literature with textual programs to visual programs. However, this phenomenon was not observed in the experiment that we conducted for studying the effect of programming paradigm in Chapter 3, nor was it observed in the literature by Curtis *et al.* (1989) and Moher *et al.* (1993). We provided here a discussion arguing that visibility overrode the 'Match-Mismatch' effect in those studies. Secondly, scrolling appeared to affect response time more than accuracy in our experiments. However, we also observed that the effect of scrolling was not as critical as '*fall back*' and could be overridden by its effects. And finally, we found participants who were more experienced in construction toys like Lego performed better in Flow Study 2. This finding echoes that of Green & Petre (1993) that graphical readership skill; in this case – diagram reading – can be trained.

5. USABILITY EVALUATION OF A VPL

5.1 Introduction

So far we have focused on issues concerning notational design. The issues discussed and the lessons learned from the empirical studies in the previous chapters relating to programming paradigm and perceptual coding can inform the design of a VPL. However, the investigations employing the experimental method proved to be time-consuming and narrowly focused. Considering that this is a PhD research with limited resource and a short time frame, it would take far too long to achieve our research objectives. Therefore, the empirical study presented in this chapter takes a different approach. The purpose of this study is to obtain a list of problems potentially encountered by novice programmers learning VPLs for the first time, which is used in a later analysis (in Chapter 6) to produce a usability checklist and principles for VPL design as stated in our research objective statement (Chapter 1). In order to achieve this, a commercial VPL, Prograph, is evaluated holistically. Two main tasks prior to the evaluation of Prograph itself are to identify appropriate research methods and a suitable usability evaluation method.

5.2 Usability evaluation methods

This section explores and discusses usability evaluation methods for programming languages. Our review suggests that Cognitive Dimensions of Notations (CDs) is the best available method for the task (the evaluation of programming languages). Its strengths and weaknesses are subsequently contrasted. As a result, an approach is suggested to overcome its weaknesses, and to establish the research question further.

5.2.1 An overview of usability evaluation methods

Formative evaluation of an artefact informs its design. The purpose of an evaluation is to assess the artefact: how successful it is; whether the targets are met and if not, what the problems are or are likely to be. The bottom line, though, is that the stakeholders or users of the evaluation (management, designers, developers, government agencies, etc.) must be able and willing to utilise its results. The users of the evaluation not only need to know the problems but also the recommendations for improving the artefact.

In HCI, evaluation methods have been well developed, used, and tested for evaluating user interfaces. A typical method used is laboratory testing (or usability testing), a very effective method in generating a list of usability problems and recommendations, but expensive and time consuming. Hence, the method is not practical in all situations.

Usability inspection methods, such as Cognitive Walkthrough (Polson *et al.*, 1992) and heuristic evaluation (Nielsen & Molich, 1990), are alternative methods used by usability practitioners. These methods are cost-effective and particularly suitable for evaluation during the early stages in the design life cycle and are best conducted by usability engineering experts or the users who are knowledgeable in the user interface domain (Karat, 1994). However, since the heuristics used by these methods have been derived from user interface problems and interface design guidelines, which concentrate on users' interaction with the interface, these inspection methods are not entirely suitable for evaluating programming languages.

Programming activities are complex. Not only must the programmers learn to handle various programming concepts inherent in the language, they must also learn how to use the programming environment. The process of programming is iterative and exploratory (Green, 1990) and very much dependent on individual differences and pre-programming knowledge. No evaluation method devised for user interfaces is adequate for testing programming languages, with the exception of laboratory testing (to some degree). In practice, laboratory testing is usually carried out to test the usability of certain functions of a programming language, which are either of special interest or those that are frequently used. However, when evaluating a new language there are many possible features and problem areas to investigate. Using laboratory testing for the whole language would be very resource demanding. We therefore seek a cost-effective method that will provide as complete a coverage of the programming language as possible.

A few methods have been employed to evaluate programming languages (Bell, *et al.*, 1992; Yang *et al.*, 1995; Green & Petre, 1996). Most of these, however, do not provide complete coverage. Bell *et al.* (1992), for example, used the Cognitive Walkthrough method to evaluate the "writability" of the features of a programming language that its designers are interested to know. However, it was reported that evaluation results of this method are dependent on the exercises planned for the evaluation (Bell *et al.*, 1992).

Yang and his colleagues (Yang, *et al.*, 1995) have developed a design benchmark for VPL navigable static representation. The benchmark provides designers with concrete measures such as the number of steps required for navigation to achieve certain tasks. However, it only applies to navigable static representation in VPLs.

Green (1989) proposes a set of cognitive heuristics, which he calls Cognitive Dimensions of Notations (CDs), to be used by non-HCI experts as a broad-brushstroke framework for evaluating usability of information artefacts. He and a colleague (Green & Petre, 1996) demonstrated that it could be used to evaluate VPLs. This framework has much potential as a usability evaluation method for programming languages as will be subsequently discussed. From here on we shall refer to the method as 'CDs'.

Like any other inspection method, the procedure to carry out an evaluation with CDs is rather broad and has room for improvement. For the time being, evaluators can use the method in two ways: a) by conducting a CDs analysis and b) by using the CDs Questionnaire designed by Blackwell & Green (2000). In the former, the evaluator looks for something in the artefact that would violate any of the dimensions in the CDs. In the latter, the users fill in a standard CDs questionnaire (Appendix C-1) and return it to the researcher/evaluator, who then conducts both quantitative and qualitative data analysis from the data.

5.2.2 The Cognitive Dimensions of Notations (CDs)

CDs have been used by several researchers to evaluate programming languages and specification languages before (Modugno, 1996; Green & Petre, 1996; Kutar *et al.*, 2000; Cox, 2000; Clarke, 2001). At present, there are fourteen dimensions in all. The dimensions provide evaluators with a broad-brush discussion tool to evaluate the usability of their products. The dimensions and their description in relation to evaluation of programming languages have been given in Chapter 2. A discussion as to why CDs is the most suitable method for evaluating programming languages and how it might be improved is offered.

The strength of CDs

The strength of CDs lies in its breadth and depth of the coverage that the evaluation can give when resources are limited. The method can be applied to non-interactive systems as well as interactive systems. This makes it more suitable for programming languages than other usability engineering inspection methods that focus on users' interactions with interfaces such as heuristic evaluation (Nielsen & Molich, 1990). The dimensions form a checklist that reminds evaluators of potential problems arising from different aspects across the whole spectrum of programming activities. Because the dimensions focus on cognitive issues severe problems do not tend to be overlooked. The CDs analysis does not require users to perform tasks because the dimensions are used as a discussion tool by evaluators. Hence, no experiment is involved, unless empirical data is needed for confirmation. The method is analytical and cost effective. The CDs Questionnaire is a standard form designed to be generic for use with any information artefact, which is sent to users of the artefact being evaluated. The main advantage of this method is that "the users do all the work" (Blackwell & Green, 2000). However, the CDs Questionnaire is only useful if there is a pool of users of the artefact who are willing to respond to the questionnaire.

There is a consensus among programming or specification language designers that CDs is a useful evaluation method (Modugno, 1996; Kutar *et al.*, 2000; Cox, 2000; Clarke, 2001), especially when the designers themselves conduct it as it enhances their understanding of the systems and/or notations (Modugno, 1996; Kutar *et al.*, 2000). However, these opinions on the ease of use of CDs vary as further discussed below (Modugno, 1996; Kutar *et al.*, 2000; Cox, 2000).

<u>The weaknesses of CDs</u>

Since CDs is predictive some problems that could be revealed by laboratory testing may be overlooked as evidenced in Clarke's (2001) report that some problems found in laboratory testing were not revealed by the CDs Questionnaire data obtained from the same participants and vice versa. However, although there has not yet been any evidence of this with the CDs analysis, we anticipate poor overlapping of results among different evaluators or between laboratory testing and CDs analysis—a common problem for expert review methods (Chattratichart & Brodie, 2002a; Molich & Robin, 2003). CDs evaluators only speculate about problems (which can later be supported by empirical data) but there is no users' feedback or recommendation for re-design from users unless the problems are empirically supported and users' data are collected. The main procedure for the analysis of the dimensions is to "consider each notation in terms of the list of dimensions, identifying any usability problems where the system characteristics on that dimension are inappropriate to the user activity, for example, high viscosity is inappropriate to exploratory design" (Blackwell, 2000). This can be quite subjective (Wilde, 1996; Cox, 2000) and dependent on the evaluators' way of thinking, mindset, and experience. The results highly depend on how the evaluators interpret/understand the meaning of each dimension and what situations, tasks, or scenarios they have in mind at the time of evaluation. Moreover, "a baseline is lacking" (Wilde, 1996), thus evaluation outcomes of the same system but by different evaluators are not comparable.

Furthermore, although CDs is aimed at non-HCI specialists (Green & Petre, 1996), it was found to be difficult to learn and to use (Wilde, 1996; Kutar *et al.*, 2000). To ease its use, it has been suggested that the artefact be evaluated based on the dimensions in the Cognitive Dimensions Profile that is created analytically specifically for it, by focusing on target activities of the users: incrementation, transcription, modification, exploratory design, or searching (Green & Blackwell, 1998; Blackwell, 2000). Having defined CDs Profile as 'the desirability of each dimension for a specific activity' and determined the profile for the specification language that they were designing, Britton & Kutar (2001) evaluated the language based on the profile. They also conducted an empirical study that provided evidence suggesting that some dimensions that were not in the profile had been overlooked.

Contextualising the CDs framework

CDs has been well received among academics but less so in industry by usability specialists, having only been used by researchers at Microsoft, Bentley Systems, and Synquiry Technologies (Blackwell, 2002) at the time that this thesis is written. Conducting a CDs analysis can sometimes be difficult because of its lack of context and session observation. Results of the evaluation from different evaluators or responses from different users are not likely comparable. The result is as good as the interpretation of, and the scenarios for, each dimension that the evaluators have in mind. For use in an iterative design it may be best carried out by the same evaluators so that the results can be consistent because the same baseline and understanding of the dimensions can be applied throughout the life cycle.

Since CDs analysis is meant to be activity-based (Green & Blackwell, 1998) incorporating factors such as modification, transcription, but not necessarily task-specific. There is no obvious link between speculated problems and specific tasks, i.e. the technique cannot give a list of usability problems in the usability engineers' context (e.g., what tasks the users cannot do). The users or clients of an evaluation (management, developers, and designers) want to see a list of usability problems so that they can set priority to fix them. To gain wider acceptance, CDs should give its users a set of usability problems or, at least, its procedure should be made more task-oriented.

What is needed is to involve user tasks so that the analysis can be more contextual; hence the evaluation reflects real problems and, at the same time, can be done more easily because evaluators can be kept focused. It would be ideal to analyse each dimension based on all sorts of tasks required in programming. However, there are simply too many of them. In writing or producing a correct program, the programmer has to read, understand, test and debug the code. During the programming process these activities are intermingled and cannot be separated from one another. Furthermore, apart from the problems caused by hard programming concepts, there could be other problems arising from dealing with various issues such as, representation, syntax, interface, and so on.

5.2.3 Reducing the analysis space: An approach to improve CDs

At present, the way CDs analysis is conducted is that the evaluator goes through the dimensions, one by one, either with or without using the system, to do some representative tasks (writing a program, for example). For each dimension, the evaluator thinks of some scenarios/situations in which certain (programming) features could be problematic or find some problems while doing the tasks. The possibilities for analyses are unlimited. Thus the analysis space can be large. Without using the system, this approach is rather ad hoc. While one evaluator may be thinking of a low-level aspect such as at the cell level in a spreadsheet program, another may be considering problems at a higher level such as creating a macro in a word-processing application to speed up certain tasks. The problems found thus are dependent on the evaluator or the questionnaire respondent, which could vary considerably. The reliability of evaluation results by different evaluators is therefore questionable. In order to improve its reliability, we suggest that we start considering a way to reduce the analysis space in exploring each dimension.

One way to limit the analysis space is to couple the analysis with tasks. That is, CDs should be made explicitly task-oriented. By doing so, each task considers one dimension at a time. However, as mentioned before, there are just too many tasks in programming. It would be impossible to think of each dimension in terms of each trivial task. We believe that usability problems arising from tasks that users cannot do can be grouped into problem areas or categories. So instead of coupling the analysis with tasks, it may be more plausible to couple it with problem areas, thus breaking the analysis into smaller chunks without losing the whole. By examining each dimension based on each problem area, the analysis can be carried out in the same fashion repeatedly and hence more consistently. However, it is not known what and how many problem areas there could be for programming languages in general. Our question is, what are the potential usability problem areas for a learner of a new VPL? Subsequently, how does one carry out an evaluation study that will consider a VPL as a whole and not just some of its specific features?

5.3 Usability evaluation of Prograph

In order to find answers to the research question above and to obtain a list of potential problems, in this section we review research methods suitable for the evaluation and subsequently provide detail of the empirical study carried out. Prograph is used here for various reasons. It is the only commercially available general-purpose VPL according to Blackwell *et al.* (2001) and therefore has been well tested for use commercially by professional programmers. It is reasonable to expect that the most obvious usability problems have been ironed-out during its design and development process, and those that will be found during our evaluation should be worthwhile considering. By evaluating Prograph we are, in effect, conducting a competitive analysis: using an existing product as a prototype for the design of a target product (Nielsen, 1993). Results from the evaluation should therefore be more representative of real-world software development applications than results from evaluating some micro-languages devised specifically for the empirical study as commonly exercised (see, for example, Sime, *et al.*, 1977; Gilmore & Green, 1984; Good, 1999).

5.3.1 Methodology issues

At the end of Section 5.2, we asked what the *potential usability problem areas* for a learner of a new programming language could be and, subsequently, how does one carry out an evaluation study that will consider a programming language *as a whole*? Here a discussion of issues relating to research methodologies in order to find a suitable research method for the questions is raised.

Exploring the research questions

In order to select an appropriate research methodology, we explore and relate the research questions to what has already been known, what is being explored, whether further issues emanate from the original questions, and whether any hypothesis can be formed.

Usability problems in the context of programming are related to poor program comprehension. However, as we have discussed in Chapter 2, program comprehension can be enhanced if the program displays the information that is required by tasks or that is obscured well. For example, in a data flow program control flow information is obscured. In order to answer a forward question the control flow information must be supported, otherwise performance suffers. That control flow information is obscured is thus a usability problem because it makes the program difficult to comprehend. However, forward tracing is not the only programming task. There are so many possible programming tasks, each of which may require different information. The full list of what obscure the information required by all sorts of programming tasks is hence unknown. We can envisage only some of the causes, such as poor visibility of programming entities, implicit or invisible dependencies, incomprehensible comments or operator names, etc. How many more and what are they? We do not know.

Usability problems of programming languages can be caused by poor design of programming constructs. Beginner's difficulties with programming arise from negation, conditionals, transfer of control, and the context-free programming syntax which humans do not find easy (Green, 1980). As we have discussed in Chapter 2, some programming language constructs do not have a cognitive fit with the way novices work in real life. These are iterative constructs, assignments, variables, and so on. However, these are related to textual programs. Would we find the same in visual programs and would the representation of these constructs in VPLs be attributed to problems that novices might experience due to lack of cognitive fit?

There are many other questions, such as those relating to error-proneness of VPLs' visual representations and the look-and-feel of the interface. What parts of the programming language are error-prone? When do they become error-prone? How serious are the problems incurred by error-proneness? What about the look-and-feel of the representations for the constructs - do they affect how well the information is displayed? If they do, how serious is the effect? These are only some questions to problems relating to error-proneness and the look-and feel of the interface. There could be more problems but what are they? We do not know.

The experimental method

One way to answer the above questions is to take a quantitative approach using the experimental method. The advantage of this method is two-fold. Firstly, in this method, non-relevant variables can be controlled. Secondly, it employs statistical analysis of the data obtained through standardised measures of large number of samples, and therefore, facilitates comparison and generalisation of the findings.

However, there are also disadvantages. Firstly, the approach is not economically viable, as a large number of experiments need to be carried out to answer all the questions one can envisage, that will cover the whole programming language. Secondly, there are potentially many independent variables and therefore the interactions between them can become too complex to be handled and the effect of each individual variable is difficult to be interpreted correctly. Thirdly, in controlled experiments, participants perform tasks in a setting that is catered for by observing only the effect of the variables of interest. Hence other variables are held fixed. This kind of setting is unnatural and out of context of use (in this case, learning to program using the programming language in question). Finally, using the experimental method, the problem areas (our research question) would have to be pre-determined because hypotheses must be formed prior to the experiments. In this case, we must know what the problem areas are before we form the hypotheses for our experiments. However, it is impossible to know all the problem areas. Forming hypotheses is, therefore, not practical. An alternative method that does not require the problem areas to be pre-determined was sought.

Qualitative inquiry

As discussed above, the experimental method and the quantitative approach are unsuitable for the research question in focus. What is needed is an approach that is exploratory, so that problem areas are not pre-determined and would emerge naturally within the context of the research. Considering its alternative, the qualitative approach, we compare and contrast the characteristics between experimental method and qualitative inquiry as summarised from Lincoln & Guba (1985) and Patton (1986 & 1990) in Table 5.1 and Table 5.2.

Table 5.1 Characteristics of experimental method and qualitative inquiry

EXPERIMENTAL METHOD	QUALITATIVE INQUIRY
1. Focused	1. Open ended
2. Controlling and manipulating - Researcher manipulates the setting by changing the level of treatments/variables and controlling extraneous variables. - Treatments and outcomes are represented by variables. - Operational definitions (variables and their measurements) must be defined in advance. - Unstructured data are of little value.	2. Naturalistic - Researcher does not manipulate settings or control variables. - There is no notion of variables. - Research design emerges rather than being specified in advance. - Researcher takes whatever emerges as data.
3. Deductive - Hypotheses are formed in advance. - Researcher takes only data for predetermined set of variables. - Researcher attempts to confirm, or disprove his/her expectations of result.	3. Inductive - No hypothesis or constraint of outcomes is formed in advance. - Categories emerge from experience and whatever emerges from observation data or other sources available (e.g. documents, interviews). - Researcher's understanding is grounded in direct experience and participation in the setting.
4. Specifically - Understanding, if exists, is limited to what is related to the hypothesis. - Experiments are conducted in a controlled environment and hence are not contextual. - Data collection is pre-planned. Focus is given to only a few variables of interest. Results are capped within the scope of the hypotheses.	4. Wholly – Holistic - Understanding the phenomenon as a whole. - Context is vital to understanding the whole phenomenon. - Researcher obtains data from the open-ended observation and hence multiple aspects of the setting. Nuance, interdependencies, complexities, idiosyncrasies can be captured.
5. Static - An experiment is a snapshot of interested task or event. - Setting is tightly controlled. Therefore effect of changes cannot be accounted for.	5. Dynamic - Qualitative inquiry studies the phenomenon over a period of time and is not limited to a specific and predetermined event. - Qualitative inquiry assumes an ever-changing world. It expects changes, development, innovation as inevitable part of human experience. The effect of changes can be accounted for.

Table 5.1 (continued) Comparison of experimental method and qualitative inquiry

EXPERIMENTAL METHOD	**QUALITATIVE INQUIRY**
6. Generalisation - Experiments (ideally) involve a large number of participants. - Results are generalisable based on statistics. - The concepts of reliability and validity are relevant.	6. Uniqueness - Qualitative inquiry involves small number of participants, cases or events (the number can be as small as one). - Results are not generalisable but may be transferable to another similar context. - Reliability and validity are irrelevant, the concept of dependability and credibility apply.
7. Standards (measurements as instrument) - Data come from standard measurements. - Operational definitions must be defined in advance.	7. Neutrality (The researcher as instrument) - Findings come from the researcher's own interpretations. He/she takes neutrality as a stance towards his/her findings. That is, he/she enters the arena with no axe to grind, no theory to prove, no predetermined results to report.
8. Objectivity - Detachment and distance mean objectivity (unbias). - Introspection and reflection are considered subjective.	8. Empathy and insight are important - The researcher is capable to understand the feelings and experience of participants through personal contact with them, thereby gaining empathy and insights. - Qualitative inquiry emphasises the value of *verstehen* doctrine, i.e. human capacity to know and understand others through emphatic introspection and reflection (detection of emotions). - Researcher's feelings, perceptions, experiences, and insights are taken as part of the data
9. Rigid design. - Hypothesis must be formed. - Operational definitions must be defined in advance. - Once the study begins, there is no return.	9. Flexible design - Design cannot completely specified in advance. - Design develops, emerges, and unfolds naturally during the study. - Data collected during the study can be partially analysed and used to help shape the study.
10. Ideal – No notion of iterative design - If the ideal is not possible, inconclusive results may be obtained. - No iteration within the project life is possible, only repeat the study with a new design.	10. Never an ideal one – the notion of iterative design - In practice, zero manipulation is only a matter of degree. The project starts without manipulation but as it rolls, the researcher consciously works back and forth between parts and wholes, sorting out and putting back interrelated and complex variables.

Table 5.2 Advantages and disadvantages of experimental method and qualitative inquiry

EXPERIMENTAL METHOD	QUALITATIVE INQUIRY
Disadvantages - Over-simplifies the complexities of the real world. - May miss important factors that cannot be quantified. - Non-contextual → applicability to real world is questionable. - Static snapshots of an event → changes are unaccounted for. - Rigid design - Setting non-representative of real world → external validity low	**Advantages** - Holistic: important factors are not missed out, no matter how large or small. - Complexities can be taken care of. - Contextual → external validity is high. - Dynamic-changes are acceptable. - Evolving design - Natural – minimum manipulation → external validity is high.
Advantages - Generalisable - Reliable + valid - Unbiased - Standardisation - Statistics - Facilitates comparisons	**Disadvantages** - Small cases - Closeness → Subjective - Difficult to generalise - May not be repeatable (poor reliability and internal validity)

From the above tables (Table 5.1 and Table 5.2), it becomes clear that a qualitative inquiry is appropriate for this present study because, firstly, in this approach the evaluator does not manipulate the situation as in controlled experiments. Themes, patterns, and categories can emerge naturally from the inquiry. Secondly, qualitative inquiry uses inductive analysis. What emerges from the inquiry is induced from the researcher's understanding of the situation and phenomenon under the study via his/her open-ended observation, the field data, and documentation collected during the observation (Patton, 1990). By either observing participants in the field or by being a participant, the researcher can gain deep understanding of the participant's experience (Kotarba and Fontana, 1984). Finally and most importantly, the qualitative approach is holistic. The researcher understands the phenomenon as a whole (Patton, 1990). This serves our purpose of evaluating the VPL in whole, not in part. Conducting a qualitative inquiry in studying novices learning a VPL of interest will therefore allow the problem areas to emerge naturally within the relevant use context. The problem areas that emerge should cover the whole spectrum of the VPL.

Practical considerations for research design

In the previous section, we decided that a qualitative inquiry for this present study should be conducted. However, there is no definite way to carry out such an inquiry. As already stated in Table 5.1, its design is flexible, "develops, emerges, and unfolds naturally". This section is therefore devoted to discussing how the inquiry should be carried out, given the resources that are available.

In designing the research, the following essential characteristics of a qualitative inquiry must be strictly adhered to:

1. Getting close to data.

Data come from participants. In effect, we must get close to the participants to be able to develop empathy, insights, and understanding of the participants' whole experience.

2. The inquiry must be naturalistic and contextual.

To be naturalistic it is important that the inquiry is open-ended. That is, participants should not be assigned to pre-determined tasks because this can keep them focussed only on what the researcher might be interested in or anticipate. Therefore some categories (usability problem areas in this context) could be missed. To be contextual the inquiry must occur in the real situation where participants learn the programming language.

3. The inquiry must be holistic.

Again, this means going into the field studying the whole phenomenon, not just snap shots of particular events as in an experiment. In other words, we must study the participants' whole experience of the learning process from knowing nothing about Prograph to being able to program in the language, not just designing some programming tasks for participants to do, which capture only '*parts*', not '*whole*'.

4. The inquiry must be dynamic in nature.

Changes are expected and allowed in this methodology. Therefore, the study has to be carried out over a length of time to cater for changes resulting from gradual understanding and familiarity of the features and concepts of the programming language during the learning process.

The above requirements, drawing on the essence of qualitative inquiry, bring up the issues of the research setting, methods, and data acquisition. The setting has to be as natural as possible. It has to mimic students or novices learning the language from anew and over a period of time until the language is mastered. For the latter issue, a decision on appropriate methods and data collection techniques to be employed in the study must be made.

Now, let us consider the resource available to carry out the present study on the two issues identified above.

1. Settings that mimic the learning of the Prograph language.

Prograph is not a teaching language. Therefore it was not possible to find Prograph student programmers to participate in the study. An option is to teach the language to some student volunteers. However, finding volunteers to spend months to master the language was not possible. Nor was recruiting paid volunteers a viable option. The only solution was to use the researcher herself to both learn and evaluate the programming language. The researcher had never learned any VPL before, was not a professional programmer, and did not have to be paid for the study.

2. What methods and data collection techniques should be used?

It was decided above that this study would use the researcher as the learner participant of Prograph. This imposes a question of whether an inquiry can be carried out using *'self'* and how to ensure credibility of such a study. Upon investigation of several research methods and techniques, we identified the most applicable methods to the constraints and objectives of this present study. These are: participant observation, self-observation, immersion, and diary studies.

Methods and techniques used

The following sections describe each method and discuss its appropriateness to this study.

Participant observation

Participant observation is considered a research strategy or method to gain access to the human experience from the insiders' view and is generally practiced as a case study. It requires that the researcher become directly involved as a participant himself/herself so that he/she does not only *see* what is happening but also *feels* what the experience is like. Firsthand experience gained through participant observation can be an extremely valuable resource of data because it promotes *verstehen* (understanding) and hence, empathy and insights (Jorgensen, 1989).

The main source of data in this technique is a collection of field notes. Whilst benefiting from firsthand experience and understanding, the participant observer's field notes can be biased due to his/her personal involvement. For the research to be credible, Bruyn's (1966) suggestion should be taken seriously: that the role of participant observer requires both detachment and personal involvement to deal with the interdependence between the observer and what is observed by developing a strategy that will allow the "observer to experience the phenomenon being observed, while at the same time maintaining sufficient separation from the phenomenon to permit the observer to be an observer—to abstract the experience and the phenomenon" (Patton, 1990).

Self-observation (Auto-observation)

This method is a variation of participant observation in that the researcher is the participant observer observing himself/herself in natural settings. The use of self as a research tool is rooted in the notion of *reciprocity of perspective*: "that people can see the world from the eyes of others, in assuming that people experience similar feelings and emotions in reacting to the world around them" (Adler & Adler, 1994). Self-observation is a method used mostly by existential sociologists who maintain that, "one must immerse oneself in everyday reality – feel it, touch it, hear it, and see it – in order to understand it" (Kotarba & Fontana, 1984). In this method the researcher's experience of self becomes data for the inquiry and serves four purposes in developing an understanding of the phenomenon:

1. Experience is a firsthand source of data. This is especially crucial for discerning the hidden aspects of human reality.
2. One's experience provides a basis for comparison with the experiences of others.
3. One's experience generates points of inquiry.
4. One's experience helps the researcher attain a theoretical understanding of real events. The participant observer who operates with good faith and realises the complexities he himself faces in making sense of the world is reluctant to espouse unrealistic and simplistic explanations for other people's behaviour. (Kotarba, 1977)

In this present study, where the researcher is the only participant available, she would have to take a complete membership role in the observation – being both evaluator and the learner of the VPL. In self-observation, the researcher's data source is usually a narrative text chronicle written in diary form (Adler & Adler, 1994). The diary data (field notes) have been used to provide insightful data and/or for introspection process (actively thinking about one's thoughts and feelings) as shown by the following examples.

Self-observation data as an insightful source of data

Adler (1984), who took a participant role as a coach to a college basketball team and became a celebrity by chance, observed himself becoming a celebrity and how his celebrity role affected his data gathering and his understanding of the team members. His self-observation proved useful to his inquiry.

Self-observation data used for introspection

Introspection can be achieved in dialogues with self and others or by reading and analysing other's free writing—non-stop writing about what they are thinking and feeling and what it means to them (Ellis, 1991). Ellis (1991) shows, from her findings of four studies, one of which was self-introspection, that introspection can generate interpretive materials from self and others that are useful for understanding the lived-experience of emotions. After all, she argues, "Who knows better the right questions to ask than a social scientist who has lived through the experience? Who would make a better subject than a researcher consumed by wanting to figure it all out?" (Ellis, 1991).

Introspection gives good insights

Krieger (1985), who took an active membership role to studying a lesbian community, collected a large number of interview data and field notes. However, she was unable to generate any useful interpretation from her data for a full year because she felt that she was not distant from the data enough and that her own feeling and experience with the members of the community interfered with her interpretation of the data. Only after she resorted to introspection by conducting systematic dialogues with herself about the experience of her involvement with the community and in conducting the interviews could she come to understand the lesbian community she was involved in and hence interpret the data. The introspection gave good insights to her analysis of the interview data.

Immersion

Immersion technique has its root in sociology, which has been discussed in the above section. It has become increasingly used as a method for understanding user requirements in product design (Jordan, 2000). In this technique, the designer lives the user's experience. While 'traditional' user-research methods tend to observe people from the outside, immersion is about trying to live as the user would, use the products and services the user would use and really get inside the user's skin. Moore (1985), a leading proponent in Universal Design spent three years in her twenties living the life of an 85-year-old woman travelling 116 cities all over the America with her joints bound to simulate the effects of arthritis. This is probably the most famous and extreme example of the use of this technique. The insights gained about the problems that older people have with a whole range of products and services have served as valuable input to a whole range of designs ever since. Similar approaches, although far less extensive in terms of time, have been used for understanding the experience of disability, including the 'disability suit' developed by Loughborough University in the UK which simulates a variety of mobility problems for the wearer (Hitchcock & Taylor, 2003). The Royal National Institute for the Blind has also developed a series of glasses to simulate visual impairment and institutions representing deaf people have devised systems that simulate hearing impairments. Meanwhile it has become standard practice for many physical rehabilitation courses to insist that their students spend some time in a wheelchair in order to get an understanding of some of the issues that their patients face. In health and well-being products, a number of design consultancies have used immersion as the basis for user-research method for the design— including heart rate monitors, blood-pressure meters and products for people with diabetes and other conditions. This has enabled designers to gain a strong empathy with the users. The rich understanding of users that this gives has led to user-centred design insights which would almost certainly not have been uncovered using traditional user-centred design methods.

Diary studies

Psychologist, Breakwell (2000) defines a diary study as "any data collection strategy, which entails getting respondents to provide information linked to a temporal framework", i.e. it refers to the recording of information "in relation to the passage of time". This technique has been used by researchers in many other disciplines ranging from history, social science, anthropology, market research, to HCI and CSCW (Palen & Salzman, 2002; Corti, 2002). In HCI community, this technique started to gain recognition in the 1990s (see for example Kirakowski & Corbett, 1990; Chin *et al.*, 1992; Sellen, 1994; and Rieman, 1993) and is drawing more and more attention from CSCW researchers lately (see for example, Adler *et al.*, 1998; Brown *et al.*, 2000; O'hara & Perry, 2001; and Palen & Salzman, 2002).

In a typical diary study respondents (or participants) are asked to record information relating to some particular activities that the researcher is interested in, onto a medium (the diary), as regularly as possible for a period of time. The medium can be of any sort: paper, electronic documents, photographs, or even voice messages. The researcher's role during the study is to provide a point of contact to answer to any queries or deal with any problem that should arise and to keep in touch with participants in order to encourage regular recording.

Major advantages of diary studies are familiarity, intimacy, and sequencing of data. People are usually familiar with the notion of diary and use diary in their everyday life. Therefore, it is not difficult to explain to the participants what is expected of them from the researcher. The data obtained from a completed diary also provide sequence of events, which is an added dimension inaccessible to data obtained from the experiment method. Furthermore, there is a belief that "iterative self-reporting will engender self-revelation and honesty" (Breakwell, 2000). Therefore, a diary is an effective means to capture intimate information, not easily accessible by interviews, questionnaires, or direct observation.

As any other technique, diary study has its limitations, the most crucial being its lack of control. Major problems are under-reporting, over-reporting, or selective reporting by participants because they affect the veracity of data. Therefore, the truthfulness and the completeness of the information obtained from participants cannot be ascertained.

Diaries vary from highly structured to totally unstructured (free-writing). A structured diary consists of entries grouped into pre-defined categories of activities for participants to check off or to fill (or to answer to, in case of voice messages). Proponents to the structured approach argue that it allows for both control and context, which are not simultaneously possible in methods at either end of methodology continuum (Rieman, 1993). An unstructured diary, on the other hand, lets participants record or write anything freely. It is usually in the form of a personal or private diary. While the content of a structured diary is divided into sections of pre-determined categories, the content of an unstructured diary is thick, narrative, and non-deterministic. Structured diaries are thus suitable for research situations when categories can be pre-defined and where confirmation to some existing knowledge is its purpose. Unstructured diaries are suitable when the research is of an exploratory and discovery nature and hence, where respondents must not be pigeonholed into recording only some pre-determined category of information.

Research design

This brief section summarises the above considerations on methods, techniques, and resources to come up with a design for the evaluation of Prograph in the subsequent section. We conclude that it is best and theoretically sound to conduct the evaluation using the researcher of this thesis herself as both participant and observer. Immersion would be conducted using an unstructured diary as data collection tool. In short, a diary study is to be carried out. In order to gain an in-depth details of the problems that may occur when one learns a programming language, the inquiry is to be conducted while letting the learning process evolve at its own pace. Therefore, the inquiry would be open-ended and categories (usability problem areas) would emerge from the study. In this way we would experience the problems with the VPL *firsthand* and thereby gain deep understanding of the problems inherent to the VPL. The findings from the inquiry and the insights gained would then have credibility.

In order to establish credibility of the inquiry further, certain level of detachment would be maintained. Taking a naturalistic inquirer's stance, the researcher did not plan what data to look for or how she would analyse the data. However, as presented in Table 5.1, qualitative inquiry design is flexible and the data collected during the study can be partially analysed and used to help shape the study. This means it is not uncommon for the qualitative inquirer to use the data obtained and feed it back into the inquiry before the research is finished. The study departs from this slightly. We intend to let everything unfold naturally and *not* to use partial data (before the study completes) so that the emerged categories would be more credible. As far as possible, it is intended that the findings should also be triangulated with findings in the literature.

5.3.2 The Diary Study of Prograph

Objective

This is an open-ended search for potential usability problem areas that could be experienced by novices to VPLs.

Method

Design

A diary study employing Immersion technique was conducted, whereby the evaluator lived the user's experience in learning the Prograph VPL. Data was collected using the Diary technique.

User profile

The evaluation was conducted by the researcher of this thesis. She was new to visual programming although she had had some programming experience in several textual languages.

Materials

The commercial VPL Prograph, which is a data-flow and object-oriented diagrammatic VPL, was evaluated. Learning materials used in this study consisted of a textbook (Steinman & Carver, 1995) and the Prograph Tutorial Version 1.3, supplied by the vendor (Pictorius Inc.). The evaluation of class libraries was omitted because our target users were student programmers. Therefore this study focused on aspects covering programming constructs and features typically required by programming exercises commonly used in first year programming courses.

Procedure

As the user, the researcher was to learn to program in Prograph by attempting exercises in the textbook until satisfied that the task was mastered. To mimic natural learning behaviours of a student mastering a new programming language she did not pre-determine specific problem areas to investigate. Furthermore, the evaluation was carried out on a self-paced and self-studied basis without any technical support from the vendor. She took a double role as the user and the evaluator. She documented in her electronic diary the problems encountered, frustrations, appraisals, or opinions whilst the exercises were carried out. The diary can be found in Appendix C-2.

A total of twenty programs were written. Twelve were small non-OO programs, five were medium-size non-OO programs; three were OO programs with inheritance; and three were OO programs without inheritance. The implementation included a total of 35 class methods and 92 universal methods, 41 of which were appropriate for use as built-in functions or procedures. The whole learning period took 21 working days (150-200 hours).

5.3.3 Content analysis

Procedure

The diary recorded the user/evaluator's learning experience of Prograph, starting on February 12 and ending on March 19, 2001. The time spent during this period split between reading the texts and teaching materials and doing the programming exercises. Recording was carried out only during the programming period, in which time the user/evaluator was experiencing with using the programming language in practice. This period was 21 days long in total. The content in the diary was a list of 95 negative comments and 11 comments that were not a usability problem such as misspelling.

The negative comments were read in a chronological order and broken up into problem tokens. Each problem token was assigned a unique token number and a problem identification number (Problem ID). The token number was given in the order that it was reported in the diary. Each Problem ID was given based on the characteristic of the problem. For example, the first sentence in the transcript which said, "When there are many windows on the screen, only the active window has text description of the window on the title bar" referred to two separate problems in the first and second parts. Therefore it was broken up into two problem tokens: problem token 1 - there are many windows on the screen and problem token 2 - only the active window has text description of the window on the title bar. These two tokens were then assigned a Problem ID according to their contexts.

The first problem token above (problem token 1) referred to the number of windows on the user's screen at the time which was being a problem to her work. So an ID was given as Win 1 – Win refers to window and the number 1 in the Win 1 was given merely because it was the first kind of problem relating to windows came across during content analysis. Win 1 hence was described as problems relating to having too many windows opened up on the screen.

The latter problem token above (problem token 2) referred to the fact that the user/evaluator wanted to know what the code in other windows (inactive windows) were about so that she would know which one to click (activate) to see the code but all inactive windows had no text description on them. So this problem token was assigned Problem ID, Win 2, for windows being obscure in its functionality.

The above-described procedure was repeated throughout the diary chronologically. For every token, the previous Problem ID's were first considered whether any of them fit the problem token being considered. If it was the case then it was assigned the ID's that fit it, otherwise a new ID was created with a description appropriate to the context of the token.

When this process ended, usability problem areas emerged from the Problem ID's that had been assigned. For example, there were seven Problem ID's that associated with windows (i.e. prefixed with a 'Win'). These were then grouped under 'windows' problem area. Likewise, other problem areas such as Control Flow and OOP emerged in the same way as described. The association between Problem ID and problem area can be found in columns 1 and 2 of Table 5.3.

Results

There were 145 problem tokens (Appendix C-2) in total. These are considered usability problems because they indicate her frustration, errors, dissatisfaction, or wish for some features not provided by the language. These are the problems that will be used in Chapter 6, together with all the findings and derivations in previous chapters. This chapter focuses on the research question about problem areas as discussed in Section 5.2.3 and demonstrates how we arrived at ten usability problem areas (listed below) and the application of these findings to context other than Prograph.

The problem areas emerged from the diary study are the following:

1. Control flow

This includes comments related to the implementation of iteration and selection. They include problems with control flow representation and its syntax. Examples are:

> "How do you pass back control to another case?"
>
> "I am still struggling with loops!"

2. Graphical representation of objects

This includes comments related to certain components of objects such as icons, connecting lines, connection ports, labels of objects, and so on. Examples are:

> "The 'get' operation...the left root is not linked to anything else (only in this particular case), so why is it there? OK, it is supposed to mean that the instance is obtained and passed through the get operation, but this is not obvious."

> "Should the class method and the universal method have same or different icons? O-Oh! they are actually different.... The icon representing class method, Car, is 2-dimensional whereas the one for universals of Car is 3-D. This says, the difference is hardly noticeable, at least not by me after about two weeks of Prograph."

3. Object-oriented features

This includes comments related to implementation of object-oriented aspects of the program. A comment can refer to graphical representation of OO objects such as instances, classes, attributes, and methods; representation and implementation of inheritance; or method referencing. Examples are:

> "Subclass can't use method of parent class." (This problem was due to the fact that child class was created before the parent class.)
> "When working with objects, classes, inheritance, polymorphism, occasionally, I needed to see the 'class method' windows (both parent and children) quite often because I couldn't remember whether the method I wanted to use at the time was in the parent class or the child class....it would be nice to reserve an area on one side of the screen for easy access to whichever windows are essential."

4. Windows and views

This includes comments related to layout of the windows and window management, e.g. how easy it is to differentiate between any two windows or to tell what the code in the window represents, and how easy it is to find a particular window. Examples are:

> "When there are many windows on the screen, only the active window has text description of the window on the title bar....I often get lost, wondering where I am, particularly when the active window is down the hierarchy."
> "..and it is very hard to implement when the screen is in such a big mess!"

5. Mapping to known languages

This includes comments that indicate a desire for a feature commonly provided by other languages but not provided by Prograph or that the user could not find at the time. Examples are:

"Couldn't find the feature that will END the program in the middle of everything else like in VB."

"What is a primitive for simple 'assignment'? There is only the 'set' operation to set attribute values but not for variables because *there is no concept of variable in data flow programming!* Maybe I look for it because I am influenced by my control flow experience."

6. Direct manipulation

This includes comments related to actions on and responses from direct manipulation of graphical objects, including a desire for an icon as a shortcut. Examples are:

"I always double-click the method name to open the method window. But it doesn't. Double-clicking lets one rename the method name. To open the method window, one has to double-click the method icon!"

"When trying to create another terminal and if it is too close to the existing one, Prograph gives an error message that it's too close. Why doesn't it just stretch the icon automatically and add a terminal without giving the error message? It is a nuisance. Actually, Prograph does do it for you automatically but only when you click far enough..."

7. Help

This includes comments made while consulting the Help file, excluding typing mistakes. The comments refer to whether the information can be found or not; whether it is comprehensible or not; or whether it is correct or not. Examples are:

"The information for the primitive 'ask' gives two incorrect pieces of information: a) that there are Cancel and OK buttons but in actual fact there is only OK button; b) it references 'accept' but I could not use the primitive!"

"The stuff in the HELP-User Guide is different from what is actually available ... the list in the User Guide is different from what I have."

8. Bugs

This includes comments made regarding inappropriate system behaviour, something that can be represented but cannot be implemented, or something that can be implemented but should not be. Examples are:

> "Subclass can't use method of parent class." (Here, the subclass could be created before the parent class but the parent class's method could not be inherited)
>
> "An Initialization method is always given the name <<>> by Prograph editor."(from HELP) So why does Prograph allow me to edit a name in the <<>> ? The program worked even if I mistyped the name of the initialisation method..."

9. Error messages

This includes comments about error messages received: whether helpful and noticeable. Examples are:

> "Error messages in the bottom bar are rather difficult to understand."
>
> "When I tried to use 'accept' it gave the following msg: ... This msg is incomprehensible."

10. Harmful automatic features

This includes comments on automatic features, which are seemingly good to have, but which, unfortunately, easily cause slips such as:

> "When in Windows/View by Name mode, Prograph automatically re-arranges the icons in method windows alphabetically ...I often found it a potential source of (slight) delay and error. This was because I didn't notice the newly created/edited icon had been moved to another location."

Table 5.3 Statistical data of Prograph usability problems

Category	Problem ID	Problem Description	Counts		%
			Ind.	Sum	Cat.
Control flow	CF-1	How to pass the control /a Do Case way	10	29	20
	CF-2	Meaningless case name	1		
	CF-3	Fail, terminate, success	7		
	CF-4	Ticks and crosses	4		
	CF-5	Iteration is hard and trying to figure out	3		
	CF-6	Slips: representations	2		
	CF-7	Restrictive	2		
Icons/ Represen-tations	I-1	Seemingly redundant part	3	27	19
	I-2	Obscure meaning	5		
	I-3	Intuitiveness/ distinctiveness	4		
	I-4	Naming of operations	5		
	I-5	Mistakes cannot be easily corrected with names	1		
	I-6	Its look restricts programming style.	1		
	I-7	'Not equal' sign unconventional	1		
	I-8	Representation of program causes	1		
	I-9	Desirables	3		
	I-10	Restrictive and imposing order	3		

Table 5.3 (continued) Statistical data of Prograph usability problems

Category	Problem ID	Problem Description	Counts		%
			Ind.	Cat.	Cat.
OOP	OOP-1	Method references	1	22	15
	OOP-2	Bugs in Inheritance	2		
	OOP-3	Navigational tool for class/method hierarchy needed	4		
	OOP-4	Distinction between class and method attributes	2		
	OOP-5	Distinction between class and method windows	2		
	OOP-6	Icons related	4		
	OOP-7	Direct Manipulation	3		
	OOP-8	Desirables	1		
	OOP-9	Inflexible order of doing things	1		
	OOP-10	Available features that should not be available	1		
	OOP-11	Valid features not working	1		
Windows	Win 1	Too many windows	2	18	12
	Win 2	Obscure functionality	3		
	Win 3	Cluttered screen/messy diagrams	7		
	Win 4	Group of objects can't be commented	0		
	Win 5	Required windows hard to find	3		
	Win 6	Less Abstraction needed	2		
	Win 7	Desirables	1		
Previous Lang.	SYN	Syntax	2	14	10
	MAP	Mapping to other languages	6		
	DF	Desirables	6		

Table 5.3 (continued) Statistical data of Prograph usability problems

Category	Problem ID	Problem Description	Counts		%
			Ind.	Cat.	Cat.
Direct Manipula -tion	DM-1	Clicking wrong places/get undesired results	6	12	8
	DM-2	Difficult to know how to do things	2		
	DM-3	Mistakes cannot be easily corrected	1		
	DM-4	Annoying behaviour, misc.	1		
	DM-5	Desirables	2		
Help	H-1	Information not found or incorrect	10	10	7
		Typos	21		
Bugs	AF-1	Available features that should not be available	1	7	5
	AF-2	Available features cannot be implemented	3		
	Bugs		3		
Error messages	E-M	Incomprehensible	4	4	3
		Typos	1		
Harmful automatic features	HAF		2	2	1
Positive findings	Pos-1	Providing a list of methods	1		
	Pos -2	Alternative way to representation of math. Equation provided ... (Evaluation), ... less messy	1		
	Pos -3	Creating method on a fly	2		
	Pos -4	Inject is good.	1		
	Pos -5	Dummy method, fill in code later	1		
	Pos -6	Less typing/less errors	1		
	Pos -7	Comment of Case's visible if required	1		
	Pos -8	Ability to comment any where and hide it	1		

Table 5.3 (continued) Statistical data of Prograph usability problems

Category	Problem ID	Problem Description	Counts		%
			Ind.	Cat.	Cat.
	Pos –9	Help is easily accessed and context sensitive.	1		
	Pos -10	Good list processing capability	1		
	Pos -11	Symbol for Initialisation method stood out	1		
	Pos -12	Useful feature: link tidy	1		
		Total negative findings	145	92%	
		Total Positive Findings	13	8%	
		Total comments	158	100%	

5.3.4 Data Analyses

In this section the problems are categorised further based on the dimensions in CDs and data analyses are carried out both by problem area and by the dimensions in CDs. First, a metric for problem severity is defined and used to analyse the transcript. The severity of each problem area and each dimension of CDs is subsequently estimated from the data in the transcript. Pareto analyses are then conducted on these two dimensions: problem areas and the dimensions in CDs.

Usability metrics

Severity rating is important in that it helps usability engineers prioritise the problems found so that they can advise their clients to focus on severe problems rather than trivial problems. In usability engineering, severity rating can be done in many ways (Nielsen, 1993) as follows:

- The number or proportion of users experiencing the problem.
- Impact of the problem on the user who experiences the problem.
- How persistent the problem is or will be.

Ideally, a combination of all the above from empirical data should be used to form a severity metric. However, when data is not available, it is a common practice in usability engineering that opinions or estimates by usability specialists are sought. Since these estimates are either with or without actually using the system it has been recommended that three or four specialists be used in order to gain sufficient reliability of the results (Nielsen, 1993).

The evaluation procedure employed here involved only one user and it was not conducted in controlled experimental conditions. There was no data generated from a quantitative measurement that could be used directly as a severity metric. The only product from which some data could be derived was the transcript that the user wrote while coding over a period of 21 days. After examining the transcript carefully, there was an indication that some problems did persist. For example, the researcher, as the user, complained about loop implementation repeatedly throughout, more frequently during the first two weeks. It was then possible to at least estimate the severity of a problem by the frequency that was reported. Hence, the severity metric is defined as the frequency of problems found in a problem area or of a dimension in the CDs being violated.

Pareto Analysis

Pareto analysis is a technique widely used in industry for decision-making based on Pareto principle. The Pareto principle, devised by an economist and political sociologist Wilfredo Pareto, states that 80% of the problems are due to 20% of the possible causes ("Statistical thinking tools", n.d.). In other words, most of the problems are caused by only a 'significant few' possible causes and therefore, by correcting these 'significant few' causes, most of the problems will be taken care of ("Pareto analysis", n.d.). This practical approach helps researcher, business analysts and decision makers focus their efforts on only key causes of problems to gain optimal return for their efforts when there are too many possible actions that compete for their attention ("Pareto analysis - Selecting the most important changes to make", n.d.). For example, in business context, it means the majority of (or 80% of) the potential business values can be achieved from a few important efforts (or 20% of the effort). Therefore, a decision is then made to focus on only business activities relating to these significant few efforts. In the context of this diary study, it helps us focus on only important usability problem areas or dimensions in CDs.

The procedure below sets out how to conduct a Pareto analysis ("Pareto analysis", n.d.).

1. Tabulate the frequency data (%) of the causes in a descending order – highest to lowest.

2. Calculate and enter the cumulative frequency data for these causes in a different column.

3. Plot a bar graph with the X-axis representing the causes and the Y-axis on the left-hand-side of the graph, representing % frequency of the corresponding cause.

4. Plot a line graph on the same graph but with the Y- axis on the right-hand-side of the graph, representing cumulative % frequency of the corresponding cause.
5. Draw a horizontal line from 80% on the Y-axis on the right-hand-side to intersect the line graph.
6. From the intersection point in 5, draw a vertical line to the X-axis. This line separates the important causes (all the causes to the left-hand-side of the vertical line) from the trivial ones (all the causes to the right-hand-side of the vertical line).

In the analyses that follow, the number of comments made in each problem category or in each dimension were counted and its severity was calculated as a percentage of the total number of negative comments made.

Analysis by problem area

The 145 negative comments were put into ten problem areas described in Section 5.3.2. The data for the frequency of each problem area is tabulated in Table 5.4.

Table 5.4 Severity statistics for problems by problem category

Description of Problems	Key	Severity (%)	Cumulative frequency (%)
Control flow	Cf	20.0	20.0
Representation of objects	Rep	18.6	38.6
Object-oriented features	OO	15.1	53.7
Windows	Win	12.4	66.1
Mapping to previous languages	Map	9.7	75.8
Direct manipulation	Dm	8.3	84.1
Help excluding typos	Hp	6.9	91.0
Bugs	Bug	4.8	95.8
Error messages excluding typos	Erm	2.8	98.6
Harmful Automatic Features	Harm	1.4	100.0

Analysis by Cognitive Dimensions

The transcript was re-analysed based on the fourteen dimensions in Green & Blackwell (1998). The dimensions are: Abstraction gradient; Closeness of mapping; Consistency; Diffuseness; Error-proness; Hard mental operation; Hidden dependency; Premature commitment; Progressive evaluation; Provisionality; Role expressiveness; Secondary notation; Viscosity; and Visibility. Each comment in the transcript was assigned appropriate dimension(s). However, there were two groups of comments that did not fit any of the dimensions. The first group was of the 'I wish there was a...' type. We call this group 'Desirables'. The comments in the second group indicated that it was hard to figure out how to manipulate a certain icon or how to use a certain feature of the language. We call this group, 'Affordance'. Unlike 'Affordance', 'Desirables' are not relevant to cognitive issues because they are usually due to incompleteness of the language or to the user's knowledge of other languages. Therefore, only 'Affordance' was included in the CDs analysis. However, this inclusion was merely for the purpose of completeness of our analyses. Eighty-six percent of the problems were associated with the CDs and fourteen percent were due to the incompleteness of the language.

After assigning appropriate dimension(s) to each comment in the transcript, we calculated the severity of each dimension from the total number of comments in the same dimension. The severity of each dimension is tabulated in Table 5.5.

Table 5.5 Severity statistics for problems by Cognitive Dimensions

Dimension	Key	Severity (%)	Cumulative Frequency (%)
Consistency	Cons	17.0	17.0
Error-proness	Errp	17.0	34.0
Role expressiveness	Role	17.0	51.0
Visibility and juxtaposition	Viju	10.9	61.9
Closeness of mapping	Clos	6.8	68.7
Hard mental operation	Hmos	6.8	75.5
Affordance	Aff	6.1	81.6
Secondary notation	Secn	5.4	87.0
Premature commitment	Prem	4.1	91.1
Viscosity	Visc	3.4	94.5
Hidden dependencies	Hidd	2.0	96.5
Abstraction gradients	Abst	1.4	97.9
Diffuseness	Diff	1.4	99.3
Provisionality	Prov	0.7	100.0

Further analyses

Pareto analyses were conducted on two dimensions: problem areas and the dimensions in CDs. Results were used from the analyses to identify the set of significant problem areas worth considering and the CDs Profile for the language. Pareto analysis revealed six problem areas that should be given high priority. These are: Control flow; Representation of objects; Object-oriented features; Windows and views; Mapping to known languages; and direct manipulation (see Figure 5.1). The cumulative frequency of these problem areas is 84%. Although this figure is slightly above 80%, we include Direct manipulation because we believe that its impact would have been greater than Mapping to known languages had the researcher, as the user, been a novice in textual programming as well.

Pareto analysis of Cognitive Dimensions revealed six high priority dimensions. These are: Consistency; Error-proneness; Role expressiveness; Visibility and juxtaposition; Closeness of mapping; and Hard mental operation (see Figure 5.2). These dimensions constitute the CDs Profile that is contextual and that should be focused upon by language designers.

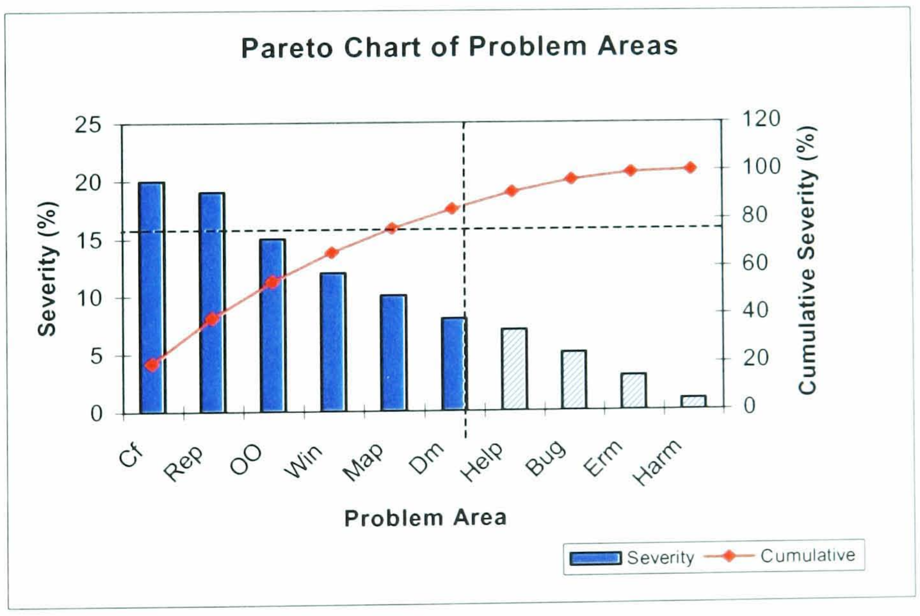


Figure 5.1 Pareto chart for Prograph problem areas

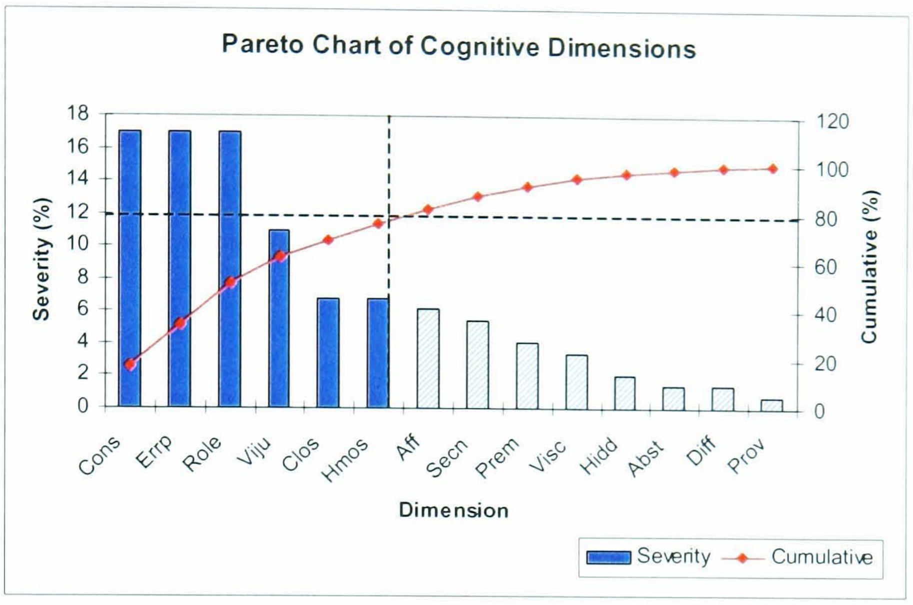


Figure 5.2 Pareto chart for Prograph violated dimensions in CDs

An extended framework for CDs

From the Pareto analyses, we have identified six usability problem areas worth investigating. These shall be called, *'Usability Problems Profile'*. The CDs Profile has also been identified, to use in future evaluations during the design life cycle. The severity data were rearranged and entered in Table 5.6 to show the breakdown of percentage of problem counts by problem areas and the dimensions in CDs. The columns and rows represent problem areas and the dimensions, in the order of their severity, from left (high severity) to right (low severity) and top (high severity) to bottom (low severity), respectively. The table is divided into four quarters. Sixty-three percent of all the problems were accounted for in the upper-left quarter, 21% in the lower-left quarter, 12% in the upper-right quarter, and only 3% in the lower-right quarter. The upper-left quarter represents 75% of the problems in '*Usability Problems Profile*'. Therefore, based on Pareto's law, it is felt that considering only the high severity dimensions with a focus on the high severity problem areas is adequate.

For CDs to be cost-effective, our suggestion, based on the two-dimensional Pareto analysis described above, is that evaluators conduct a Cognitive Dimensions analysis on the 'significant few' dimensions for the 'significant few' problem areas. The 'significant few' dimensions are: Consistency, Error-proneness, Role-expressiveness, Visibility, Closeness of mapping, and hard mental operation. The 'significant few' problem categories are Control flow, Representation, Object-oriented features, Windows and views, Mapping, and Direct manipulation. For each of the dimension, evaluators should try to answer the questions listed in Table 5.7 for each of the problem categories, which are gathered from the transcript contents.

The above leads to an extended framework for CDs. The procedure is to consider the dimensions in the CDs Profile for each of the problem areas in the *'Usability Problems Profile'*. This framework allows the CDs analysis to be contextual and the evaluator to be more focused. Hence, evaluation results can be more consistent and reliable than the original method.

Table 5.6 Severity statistics of the problem in each category for each dimension

Cognitive Dimensions	Problem Category (%)									
	Cf	Rep	OO	Win	Map	Dm	Help	Bug	Erm	Harm
Consistency		3.4	3.4				5.4	4.8		
Error-proneness	1.4	6.1	3.4			4.8				1.4
Role expressiveness	3.4	6.1	5.4	2.0						
Visibility		0.7		9.5			12%		0.7	
Closeness of mapping		2.7		63%	4.1					
Hard mental operations	6.1		0.7							
Affordance					1.4	1.4	1.4		2.0	
Secondary notation				5.4						
Premature commitment	1.4	2.7								
Viscosity		0.7	2.0	21%		0.7			3%	
Hidden dependencies			2.0							
Abstraction gradients				1.4						
Diffuseness				1.4						
Provisionality						0.7				

(The number in each cell is % Severity, and the number in a circle is the percentage of the total.)

Table 5.7 Questions to ask for the 'significant few' dimensions and problem areas

Dimension	Question
Consistency	Is the naming method consistent throughout? Is there any feature that is available but cannot be implemented? Is there any feature that is implemented but should not be made available?
Error-proneness	Naming Is the naming consistent (for example, using lower or upper case all along)? Is there any part of the name that is redundant such as brackets or quotes? Are conventional symbols (such as mathematical operators) used where they can be used? The look of the objects (icons, windows, symbols) Can it be made more distinctive or intuitive? Are there two very similar but different representations? If so, is there any effort made to differentiate them? Is the effort good enough?
Role-expressiveness	Naming Does the name tell what the icon is for? Can it be made more meaningful? The look of the objects (icons, windows, symbols) Is there any part in the icon that is redundant but does not appear so to the user? Can it be made more distinctive or intuitive? When all else fail to improve it, can comments be added to help?
Visibility	Are the diagrams messy? Can anything be done to improve them? Will there be too many windows opening at any one time? If so, is there an easy way to navigate up and down the window hierarchy? Does a new window open on top of an old one regardless of the availability of blank spaces? Is there any window that contains as little code as one graphical object or none? If so, there will be too many windows per program and therefore visibility is reduced.
Closeness of mapping	Are there any functions or features provided by other conventional languages that are not provided here, that users may ask for? Are conventional/familiar symbols and operators used? Is there any part of the representation with good closeness of mapping but redundant?
Hard mental operations	Are there any difficult concepts to learn? Can the concepts be avoided? Does the user have to think in multiple steps to use any control?

Discussion

Prograph is a data flow language. Therefore, data flow information is well represented. There was not a single negative comment in the transcript about it. On the contrary, there were many problems with its control flow representation. This finding confirms the first maxim of information representation: that some information is highlighted while others are obscured as discussed in Chapter 2.

Representation of control flow information, however, should not be the only concern in the design. Ten usability problem areas emerged from this study as potential problem areas to look for when evaluating Prograph. Rigorous data analyses allowed the extension of the CDs framework to be used when one carries out CDs analysis during the design of an improved version of Prograph. The extended framework has been derived bottom-up and is task-based and contextual. It has a solid empirical grounding in that the usability problem areas emerged naturally from the details documented by the user in the context of learning the language over a period of time.

The framework consists of a '*Usability Problems Profile*' and a CDs Profile. The problem areas augment CDs because, together, they form a pre-determined analysis space that keeps the evaluator focused while conducting the analysis. It is less likely that problems in some areas will be missed and more likely that consistent results can be obtained.

One of the problem areas in the '*Usability Problems Profile*' identified was mapping to known languages. This is both surprising and alarming. Users bring with them prior knowledge, which could interfere with their learning to use a new system. If the system or language being designed aims at a particular group of users, it would be useful for a user profile to be created for the evaluation.

The extended framework should improve the reliability of evaluation results by CDs analysis. However, one must be cautioned that using the CDs Profile may overlook some important dimensions (Britton & Kutar, 2001). Depending on the budgets and the usability goals, using the full set of CDs occasionally during the design life cycle should not be ruled out. Furthermore, these profiles are not static, they change too as we get into later stages in the life cycle. In carrying out the CDs analysis, therefore, one should bear in mind the following:

User profile is necessary.

Usability Problems Profile is desirable.

Cognitive Dimensions Profile is plausible.

Use the full set of Cognitive Dimensions if possible.

Limitations and trustworthiness of the study

There are some limitations in this study due to the qualitative approach we employed. Firstly, generalisation is not possible. But in a qualitative inquiry, one "should regard each possible generalisation only as a working hypothesis, to be tested again in the next encounter and again in the encounter after that" (Patton, 1990). The approach to the extended framework and the framework itself has yet to be tested. There are many things that need to be done. The findings need to be confirmed by laboratory experiments or by carrying out multiple case studies with different users of the same language and/or with information artefacts other than programming languages. The extended framework needs to be tested to see if it will really make CDs analysis easier to conduct and also yield more consistent results. Another limitation of this research is validity. However, the issue of validity is irrelevant here. In quantitative research, validity is gained by how accurate the measures are and whether or not the instrument measures what it is supposed to measure. In qualitative research, however, the researcher himself/herself is the instrument (Lincoln & Guba, 1985). Therefore validity depends on the researcher's personal rigour in doing the fieldwork. According to Lincoln & Guba (1985), findings and interpretations from a qualitative inquiry gains trustworthiness through *credibility*, *transferability*, and *dependability*, as opposed to internal validity, external validity, and reliability in the quantitative-experimental method, respectively. The credibility and strength of any qualitative research are gained by triangulation (Patton, 1990), which is discussed in the following section. By demonstrating the credibility of this research, its dependability can also be established (Lincoln& Guba, 1985). The last element of trustworthiness of this research is transferability, which is subsequently discussed and demonstrated.

Triangulation in designs

We have incorporated methodological triangulation in the research design by using multiple research methods, both qualitative (Immersion and Diary Study) and quantitative (Pareto analysis). The qualitative inquiry approach was employed because it was suitable for exploratory research such as this one. We wanted to find possible usability problems that a learner could experience. The quantitative approach such as experimental studies would be unsuitable because the problem areas would have to be pre-determined. However, we did not know all the problem areas in advance. Nor could we assume so. Being open-ended, the qualitative approach, on the other hand, provided us with a rich set of data, which, when combined with the quantitative data analyses on problem severity, enabled us to induce the usability problem areas, resulting in the '*Usability Problems Profile*' and the CDs Profile for future evaluation.

Investigator triangulation

To avoid bias in interpretation of the diary content, one might question why problem categorisation was not carried out by other researchers (outsiders) who were not involved in the research as a form of investigator triangulation (Patton, 1990). However, categorisation by outsiders who did not immerse in the experience of learning the VPL *firsthand* is not necessarily more correct or more reliable than if it was done by the researcher who immersed in the experience and who documented that experience. We argue that content analysis of this study is best carried out by the researcher of this study herself. This is because the researcher was the *user-evaluator-and-documenter* in this study. As the documenter, the researcher wrote the content in the diary. As the user, she experienced the frustration incurred by the problems *firsthand.* Finally, as the evaluator, she took Prograph as the subject of her evaluation inquiry. Therefore, she could empathise with the *user* (and hence knowing how severe each problem was) and understand what the words in the diary meant and in what context they were better than anyone else. As such, investigator triangulation by different researchers is not a significant issue here. After all, who else would make the interpretation closer to the experiential reality than the *user-documenter* herself?

Triangulation in data analysis

Triangulation in data analysis requires that the researcher use multiple sources of data for her analysis, e.g. interview, questionnaire, etc. However, in this study the researcher was the *user*, the *evaluator*, and also the *documenter*. Multiple data sources such as questionnaire or interview would not be applicable in this case as there were no other users or evaluators to obtain data from. Credibility of this present study thus relies on triangulation in designs (previously discussed) and the triangulation of its findings with those of other usability evaluation studies in the literature. The following paragraphs provide an analysis of the findings by Houde & Sellman, (1994) and Green & Petre (1996) to compare the results of their evaluations with ours.

Houde & Sellman (1994) carried out an observation study of eight professional programmers writing a program for a simple interface, each using a different software development environment. The environments were: HyperCard™; MCL™; Serius™; Macromedia Director™; NeXTStep™; Think C™ with ResEdit™; and two other research environments from the Apple Computer's Advanced Technology Group. Green & Petre (1996) evaluated usability of Prograph by CDs analysis. In Table 5.8, we list the problems and the dimensions in CDs that they violated according to the evaluation by Green & Petre (1996). However, we also assign appropriate problem area(s), according to the problem areas obtained in our studies, to each of the violations reported, based upon the information available in their journal paper. In Table 5.9, we list the problems reported by Houde & Sellman (1994) and assign appropriate dimension(s) in CDs and problem area(s) to each problem.

Table 5.8 Usability evaluation of Prograph by Green & Petre (1996)

Dimension	Evalua-tion	Cat.	Reason
Abstraction gradient	+	OO	Methods can be created on the fly.
Closeness of mapping	+	Cf	List processing is good and makes implementation of loop easy.
Consistency	+	Irrelevant	Better than textual languages
Diffuseness	-	Win	Too many windows
Error-proness	Not - serious	Irrelevant	
Hard mental operations	-	Cf	Repeated reversals of success and failure controls
Hidden dependencies	-	OO	Cannot navigate up the call graph to find which method call which or which is called by which
Premature commitment	-	Rep	Commitment to connection , to order of creation
Progressive evaluation	+	Irrelevant	Dummy methods can be created; code can be added or changed at run time
Role expressiveness	-	Cf	The tick and cross controls
Secondary notation	-	Win	Diagrams are untidy; Cannot use layout to communicate; group of object cannot be commented.
Visibility	-	Win	Deep subroutine structure
Viscosity	-	Rep, Win	Empirically supported; Prograph was poorer than Basic.

Note: + means good and – means poor.

Table 5.9 List of problems found by Houde & Sellman (1994)

Problem description	Cat.	CDs
"Standard features such as graphical layout tools, rulers, and alignment commands were missing. (ref. MacPaint and MacDraw)"	Map	Desirables, Closeness of mapping
"It was not possible to change the original object types, or even to 'copy' the name and position properties of the original fields and "paste" them into the number fields. This work had to be repeated."	Rep, Dm	Premature commitment
"He realized that this revision implied changing the library of drawing function included in the project. While making this change, he forgot to update other parts of the program that would be affected and spent several minutes debugging"	OO	Hidden dependency
"Some referencing problems arose from names which did not evoke the items they represented."	Rep	Role expressiveness
"The Director programmer…realized that he didn't know which one (of the four fields he created in the cast window) to put where (in the stage window). They all looked the same, and their labels could not be revealed in the stage view."	Rep, Win	Consistency
"He (the HyperCard programmer) would like to simply select all four fields to change all of their text properties at once."	Dm	Desirables
"Participants could not keep track of all the components required …They forgot where program elements were, what they were called, what state they were in, and what their relationships were to other parts of the program."	Rep	Hidden dependency, Visibility, Role expressiveness
"We noticed that the current state of the program being edited was not effectively represented to users."	Rep	Role expressiveness
"The visual identity of the program and its ties to related elements were not clearly represented…It was hard to tell them apart and …"	Win	Role expressiveness
"Appropriate views were not always available...the HyperCard programmer had to frequently select graphic elements to bringing up their individual code dialog boxes to review variable names."	Rep	Visibility
"The Serious programmer …could not access them (the desired views) in the desired order."	Rep	Premature commitment
"Before editing the graphical layout view, …could not iteratively make changes in both these views easily."		Viscosity

Table 5.10 and Figure 5.3 summarises the results from these two studies (Houde & Sellman, 1994 and Green & Petre, 1996) in comparison with ours.

Table 5.10 Comparison of three different research results

	Research results		
	Ours	Houde & Sellman (1994)	Green & Petre (1996)
Number of dimensions violated	14	7	8[a] 5[b]
Problem areas found	10	5	4
Dimensions in the CDs Profile	6	4	3 [a] 3 [b]
Problem areas in the 'Usability Problems Profile'	6	5	4

a = violations - results agree with ours; b = not violations - results disagree with ours

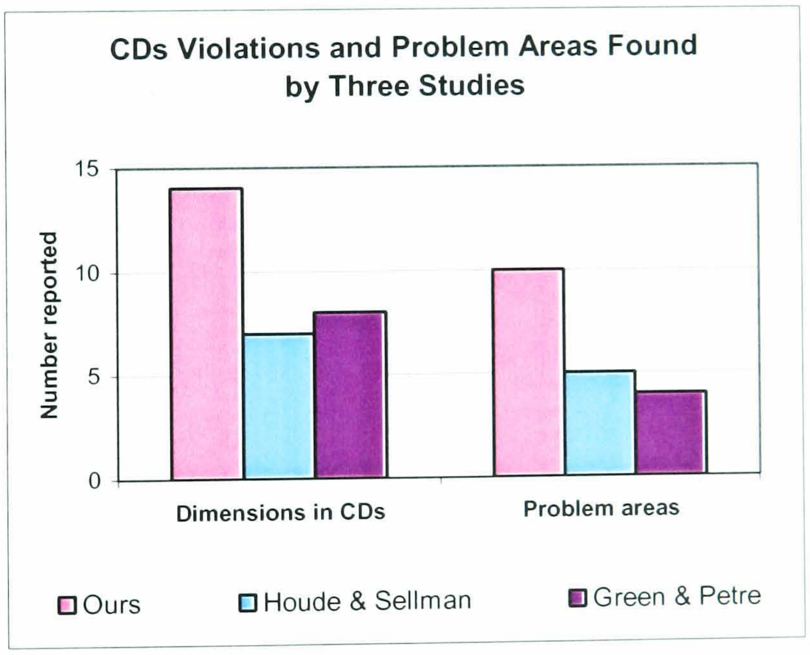


Figure 5.3 Number of dimensions and problem areas found by the three studies

According to table and figure above, our research found the highest number of problem areas and dimensions violated. All four problem areas identified by the previous research are members of the '*Usability Problems Profile*' we identified. This confirms the credibility of the interpretation of the document contents. Four out of seven dimensions violated by the first study (Houde & Sellman, 1994) are members of the CDs Profile we identified.

However, only three out of nine dimensions violated, reported by the second study (Green & Petre, 1996) are members of the CDs Profile we identified. Worse yet, three violated dimensions (Error-proneness, Consistency, and Closeness of mapping) in the CDs Profile we identified scored well by the second research. This could be because the evaluation in the second study was carried out based on only one program, supplemented with discussions with expert Prograph programmers (from email communications between the author of this thesis with Thomas Green in 2001). Therefore, some error-prone problems that are easier to discover by the evaluator(s), actually using the system, might have been overlooked. Furthermore, we found that our interpretation of the dimensions Consistency and Closeness of mapping were slightly different from theirs. We considered 'Consistency' both in their terms: 'similar semantics are expressed in similar syntactic forms' (Green & Blackwell, 1998) and ours: consistency between different parts of the system and interface. We found poor consistency in the information contained in the Help facility, although Help is not the only source of inconsistency. For Closeness of Mapping, which refers to 'closeness of representation to domain' (Green & Blackwell, 1998), we found that this dimension should include mapping features or programming concepts to/from different languages as well. These interpretations departed from the original ones but the empirical data indicated their relevance.

For the first study by Houde & Sellman (1994), however, eight programmers wrote the same program, each using a different application. The diversity of the development environments investigated and the task-based approach might have brought about more agreeable results with ours than the second research.

The similarity between results obtained by the first research and ours is quite encouraging. The two researches share one common approach, i.e. qualitative, exploratory, task-based, and contextual. It appears that sample size does not matter much and that a single case study like this one could benefit from the in-depth detail that is both contextual and holistic.

Transferability

The conventional (positivist) paradigm ("a family of philosophies characterised by an extremely positive evaluation of science and scientific method" (Reese, 1980)) assumes that findings can be generalised independently of time and context. In other words, it assumes that as long as the sample is representative of the population and high internal validity is obtainable, findings can be transferred to all contexts within the same population, i.e. generalisable (Lincoln & Guba, 1985). The alternative (naturalist) paradigm rejects this. It argues that both sending and receiving contexts need be known to ensure that findings (within some confidence limits) in one context can be applied in another. On the other hand, it assumes that the aim of the inquiry is to "form working hypotheses that describe the individual case for the next encounter and the encounter after that" (Patton, 1990). Therefore, at best, the working hypotheses can be abstracted and transferred to another similar context.

Transferability, thus, depends on the similarity between the sending and the receiving contexts that findings are transferred from and to. However, the receiving context of a future inquiry is unknown to the researcher of the inquiry at the sending end. Hence, transferability is *impossible to establish by the inquiry itself.* Transferability is therefore an empirical issue. It depends on the researcher of the inquiry to provide thick and proper description of the sending context for others to transfer their findings to another similar context.

The transferability of the approach to the extended framework for CDs we proposed above has been demonstrated by other empirical research (Chattratichart & Brodie, 2002a & 2002b; Brodie & Chattratichart, 2002; Chattratichart & Brodie, 2003; Chattratichart *et al.*, 2003). The following section describes their work.

5.4 Application of the Prograph study to other contexts

The usability problems and the problem areas derived from the diary study in this chapter is narrow in scope, i.e. it is limited to the VPL Prograph. However, the process that has been carried out is potentially useful to other research arena. The process of arriving at a set of important areas (be it usability problem areas or dimensions of CDs) for consideration during evaluation can be adopted in designing and evaluating a different application, VPL or specification language. Although findings from such an exercise should be specific to the application being evaluated they can be used, in a similar way to the way we proposed for the Prograph study, to extend the original evaluation procedure so that it is made more contextual. This extended procedure is expected to be easier, keeps evaluators more focused and therefore would result in more reliable evaluation outcomes than the original procedure.

The kernel of the extended framework proposed for Prograph in the previous section is that adding the '*Usability Problems Profile*' as another layer to the existing procedure of CDs analysis will improve the reliability of the evaluation results and, possibly, ease of use of the method. This section presents two studies of an extended method to heuristic evaluation, devised based upon the notion of '*Usability Problems Profile*' called, HE-Plus. The first study is described in detail showing that adding another layer of '*Usability Problems Profile*' to heuristic evaluation procedure improved the reliability of its evaluation results. The second study was briefly presented to demonstrate that the new procedure (HE-Plus) was easier to use than heuristic evaluation.

5.4.1 Why heuristic evaluation?

The extended framework was applied to heuristic evaluation for two reasons: 1) the similarity between the procedures to carry out the CDs analysis and heuristic evaluation, and 2) heuristic evaluation is a well-known and simple to use inspection method making it possible for us to find participants for the two studies that are subsequently described.

CDs analysis and usability inspection methods such as heuristic evaluation, guideline reviews, and standards inspections have their roots in Ravden & Johnson's (1989) methodology. In this methodology (Ravden & Johnson, 1989), the evaluators go through a checklist while carrying out realistic tasks as part of the evaluation. The tasks should be representative of typical work that users would do using the system or interface. Tasks are rated on nine criteria that are different from Nielsen's (1994) ten heuristics and the 14 dimensions in CDs (Green, 1989). Ravden & Johnson's criteria are:

1. Visual clarity
2. Consistency
3. Compatibility
4. Informative feedback
5. Explicitness
6. Appropriate functionality
7. Flexibility and control
8. Error prevention and correction
9. User guidance and support

Despite the variation in the criteria or heuristics used by different inspection methods, their procedures are quite similar. That is, evaluators perform some realistic and representative tasks and either encounter a problem or look out for points where they believe that users might have a problem based upon certain criteria, heuristics, or rules of thumb.

However, inspection methods based upon Ravden & Johnson's (1989) methodology have at least one serious problem. They are known to produce unreliable results, particularly when conducted by non-expert evaluators. Comparative studies of inspection methods (heuristic evaluation, individual and team walkthrough using 12 guidelines) and laboratory testing revealed the superiority of laboratory testing and poor overlaps in results between methods (Karat *et al.*, 1992; Karat, 1994). Poor overlap of results questions the reliability of usability evaluation methods used.

For heuristic evaluation, due to poor overlapping of findings by different evaluators, Nielsen (1993) suggests that five evaluators evaluate a product to ensure that most usability problems are revealed. The method also has other pitfalls such as false alarms and problems missed. An analysis by Bailey (2001) revealed that only 36% of all the problems identified were true usability problems while 43% were false alarms and 21% were missed when the method was used. There are three possible causes for these limitations. First, its procedure is not structured enough (Dutt *et al.*, 1994). Hence, the possible areas to be explored by the evaluator are large and results can be ad hoc. Second, the heuristics are 'often too general for detailed analysis' (Andre, 2001). Third, the set of heuristics used by evaluators may be 'faulty' (Bailey, 1999), hence the high rate of false alarms. The two empirical studies described here tackled the first two causes stated above.

5.4.2 What application to evaluate?

We need to know the list of problem areas that constitute the '*Usability Problems Profile*' for the application that we might choose for use in the evaluation but little is known about or formally recorded as a '*Usability Problems Profile*' in the literature. Our purpose is to demonstrate the applicability of the proposed extended framework. Therefore, we were free to choose the application for our evaluation. The easiest way was to avoid having to derive the profile from scratch as we did for Prograph. We believe that, for each particular type of product (be it a web site, a VPL, an online intelligent agent, a 3G interface, etc) there is a '*Usability Problems Profile*' (important problem areas) associated with it. It may be possible to approximate what these profiles are from existing research and to use them in our studies as a practical starting point.

Happily, such a profile exists for web applications, though the term '*Usability Problems Profile*' has never been used. According to Lindgaard (1994) typical usability problem categories for web sites are information content, graphics, navigation, layout, terminology, and matches with users' tasks. In the first study below usability of a web site was evaluated using heuristic evaluation method and Lindgaard's (1994) set of problem areas.

5.4.3 HE-Plus: Study 1

Objective

In this study, a between-subjects experiment was conducted to compare the reliability and ease of learning of heuristic evaluation method and HE-Plus.

Definition of terms: HE and HE-Plus evaluation methods

For the two evaluation methods used in this experiment, HE method refers to Nielsen's (Nielsen & Molich, 1990) heuristic evaluation method. The heuristics are those given in Table 5-11. HE-Plus method is what we call our extended heuristic evaluation method. In this method, evaluators performing a heuristic evaluation are given a *'Usability Problems*

Profile' to be taken into consideration on top of the heuristics used. The problem areas constituting the profile used in this experiment are listed in Table 5.12.

Table 5.11 Heuristics used in HE-Plus: Study 1

Heuristics
1. Visibility of system status 2. Match between system and the real world 3. User control and freedom 4. Consistency and standards 5. Error prevention 6. Recognition rather than recall 7. Flexibility and efficiency of use 8. Aesthetic and minimalist design 9. Help users recognise, diagnose and recover from errors 10. Help and documentation

Table 5.12 Problem areas used in HE-Plus: Study 1

Problem area
1. Information content 2. Graphics 3. Navigation 4. Layout 5. Terminology 6. Matches with users' tasks

Hypotheses

We speculated that the *'Usability problems profile'* would keep the evaluators using HE-Plus focused while conducting their evaluations. Therefore, HE-Plus group should outperform the HE group.

Hypothesis 1. The result of the HE-Plus group is more reliable than that of the HE group.

Hypothesis 2. There would be higher overlap in findings in the HE-Plus group than in the HE group.

Design

The experiment was a between-subjects design. The independent variable was evaluation method (2 levels: HE and He-Plus). The dependent variables are discussed in the 'Metrics' sub-section of the 'Results' section.

Method

Participants

Ten research students at the Department of Information Systems and Computing, Brunel University, participated in this study. All were experienced Internet users.

Materials

The web site evaluated was http://www.lakesideonline.uk.com. Each student received an instruction and training pack before the evaluation.

Procedure

The ten research students were equally divided into two groups and randomly assigned to either the HE group and the HE-Plus group. There were two male and three female participants in each group. Their task was to evaluate the usability of http://www.lakesideonline.uk.com, using either the HE or the HE-Plus method. The HE group was given Nielsen's (1994) ten heuristics to use (Table 5.11). The HE-Plus group was given the same list of heuristics and the problem areas listed in Table 5.12 (Lindgaard, 1994).

Participants were given a training pack to study a few days before the URL was given to them. The pack provided definitions of usability problems and of a problem's severity. It also included a description of the procedures for the evaluation method assigned to the owner of the pack. All packs were identical except for the information concerning the evaluation method to be used.

Participants were instructed to carry out the evaluation individually, on their own and at their own pace. They were advised to spend between one and three hours exploring the site however they wished and were required to submit a report at the end of the evaluation. The report was meant to include a description of and severity rating for each problem found, the heuristics violated, and problem areas found as applicable. Upon submission of the reports, a one-page post-hoc questionnaire was given to participants. The questionnaire asked participants to rate the web site, the evaluation method they used, and their confidence in their own evaluation results.

Results

Metrics

Kessner *et al.* (2001) compared reliability of results from usability testing performed by six usability teams in their studies with those in Molich *et al.*'s (1999) using the mean number of usability teams finding a problem. A higher mean indicates more overlap in problems found by different teams, hence, more reliable results. In addition, they reported overlap in the teams' findings as percentage of problems found by 1, 2, 3, 4, 5, and 6 teams.

Following Kessner *et al.* (2001), the mean number of evaluators finding a problem was used as a metric to compare the reliability between the two methods in our study. The percentage of problems found by 1, 2, 3, 4, 5 evaluators was used as an indicator of overlap in the evaluators' findings.

Problem grouping

A master list of problems was obtained from the evaluators' reports. The following categorisation procedure was carried out by the author of this thesis and a Ph.D. colleague. Non-usability problems were identified and usability problems were categorised independently by the two of us. A meeting was then held to resolve the 18% initial disagreement between our groupings and a final set of 36 usability problem categories was decided upon.

Data analysis

Table 5.13 presents a summary of the statistical findings of the two groups. The HE group spent an average of 3 hours on the evaluation while the HE-Plus group spent on average only 2 hours. The former group found 51 usability problems while the latter, 92 problems.

Reliability of results

The HE-Plus group yielded more reliable results than the HE group. The mean number of evaluators finding a problem in the HE-Plus group was significantly higher than that of the HE group, indicating more overlapping findings among evaluators in the former than in the latter (Mann-Whitney $z = 2.91$; $p < 0.01$).

Overlap in evaluators' findings

None of the problems was found by all five evaluators in either group. Problems found by 1, 2, 3, and 4 evaluators were 67%, 15%, 11%, and 7 % for the HE group respectively. These figures were 26%, 29%, 26%, and 19% for the HE-Plus group (see Figure 5.4).

Questionnaire results

Average ratings, on a scale of 1 to 5 (see Table 5.13) revealed that the participants found the original method easier to use and learn than the new method. They were also more confident in their own evaluations than the HE-Plus group.

Table 5.13. Results of HE-Plus: Study 1

	HE	HE-Plus
GENERAL STATISTICS:		
Average time taken (hr)	3	2
Number of problems found	49	83
Number of problem categories	27	31
OVERLAP:		
Mean number of evaluators finding a problem	1.19 (SD = 1.09)	2.06 (SD = 1.31)
SUBJECTIVE RATINGS:		
Web site experience	2.8	2.8
Usability of the method used	4.6	3.1
Confidence in own evaluation	4.8	4.1

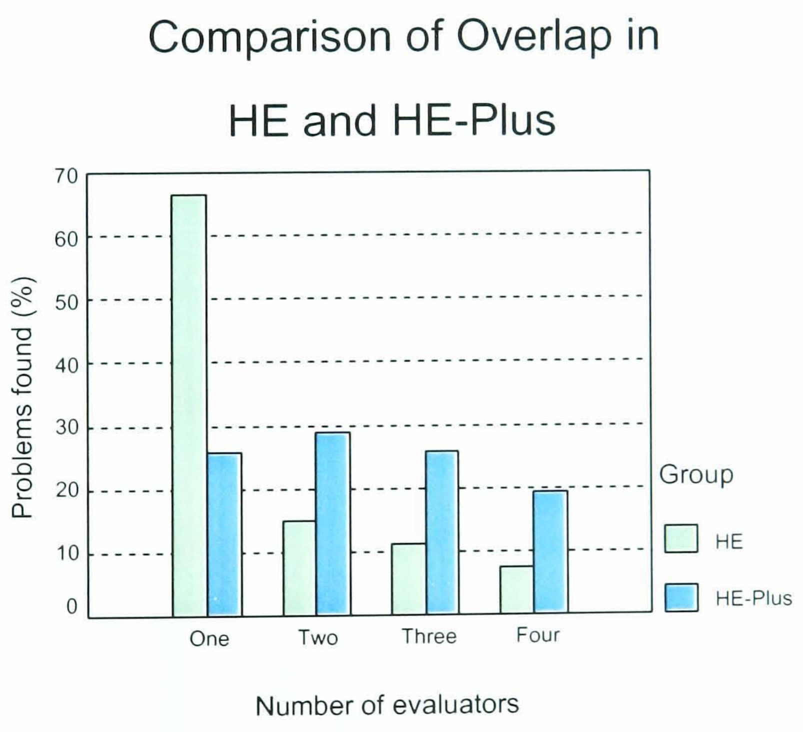


Figure 5.4 Overlap in the evaluators' findings

Further analyses

The thirty-six problem categories were further grouped according to which problem areas in the original *'Usability Problems Profile'* given to the participants. However, some problems did not fit in any of the problem areas in the profile used. The problem areas that we derived from the data are listed in Table 5.14. Note that the problems found in area, 'matches with users' tasks or what we called 'real world' in Figure 5.5 was negligible.

The percentage of problems found in each problem area was then computed and a Pareto chart was plotted in Figure 5.5. From the Pareto chart, the cumulative sum of problems found in the first 5 areas: information contents; graphics; format & layout; systems & functionality; and navigation, made up 80% of the problems found. Therefore, for lakesideonline.uk.com, these are the problem areas worth considered and focused upon.

Table 5.14 Problem areas obtained by HE-Plus: Study 1

Problem area
1. Information content
2. Graphics
3. Navigation
4. Formatting & layout
5. Systems & functionality
6. Wording (or Terminology)
7. Help & error messages

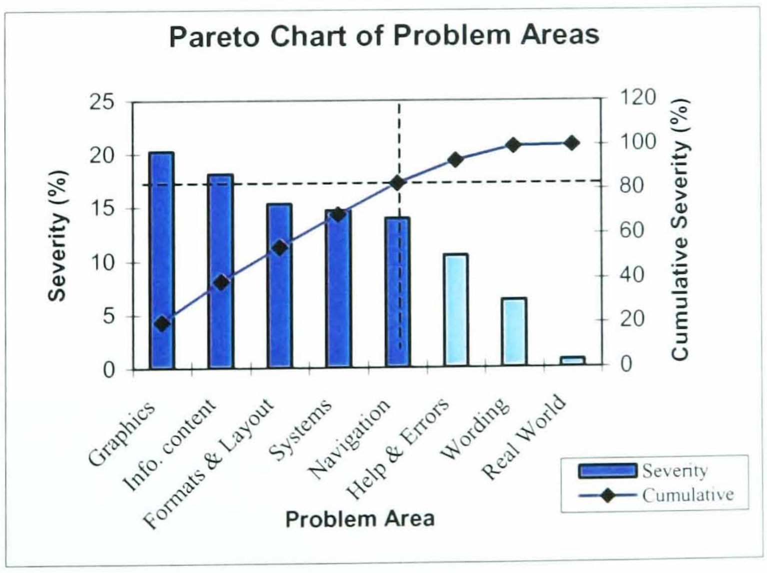


Figure 5.5. Pareto chart for lakesideonline.uk.com

Discussion

The mean number of evaluators finding a problem of the HE-Plus group was significantly higher than that of the HE group, indicating more reliable results in the former than the latter. Hypothesis 1 was hence supported. In terms of overlap, the problems found by one evaluator dropped from 67% in the HE group to 26% in the HE-Plus group. In other words, 33% of the problems reported by the HE group was found by two or more evaluators while this figure was 74% for the HE-Plus group. Hypothesis 2: that there would be more overlap in the HE-Plus group is therefore supported.

Despite the superior performance in the HE-Plus group, however, subjective ratings indicated that participants found the original method easier and therefore had higher confidence in their evaluation results. This might have been due to the additional information in the instruction pack regarding the problem areas that had to be considered, making the recommended procedure more complex for the HE-Plus group than for the HE group.

This experiment revealed that the profile consisted of only five of seven problem areas found from the data (information content, graphics, format & layout, systems & functionality, and navigation). However, it has provided evidence for the usefulness of extending the procedure to heuristic evaluation by giving evaluators a set of problem areas to focus their evaluations. The benefit of using a profile would be in its cost-effectiveness when there are many more problem areas competing for evaluators' attention (than in the case of web sites).

How do our results compare to other research?

Firstly, our novice evaluators in the HE-Plus group achieved comparable reliability results ($M = 2.06$) to the professional usability teams in Kessner *et al.*'s (2001) study ($M = 2.14$) and higher than those in Molich *et al.*'s (1999) study ($M = 1.32$, as determined by Kessner *et al.* (2001)). Moreover, the HE-Plus group had slightly higher overlapping results than those found in Kessner *et al.*'s (2001) study. In that study, 56% of the problems were found by two or more evaluators while 74% were found by the HE-Plus group in our study. The present study adds yet more weight to Kessner *et al.*'s (2001) plea for a *'focus'* in carrying out usability evaluations to achieve more consistent results.

Secondly, there is much common ground between the problem areas in our derived profile and the criteria used for rating award-winning web sites by the Webby Awards (http://www.webbyawards.com). The Webby judges rate web sites on six criteria: content, structure and navigation, visual design, functionality, interactivity, and overall experience. Sinha *et al.*'s (2001) detailed analysis of the Webby Awards 2000 dataset suggests that ratings of the first five criteria can predict the overall experience of a web site (the last criterion) and that “there are factors beyond these 5 criteria that ultimately determine award-winning sites”. Four of the problem areas in our *'Usability Problems Profile'* are shared by the Webby criteria. These are content, graphics, system efficiency and functionality, and navigation. The implications of this are twofold. Firstly, fixing problems in these four areas should help improve user’s experience of a web site. Secondly, the other problem areas in our profile - Formatting and Layout, Help and Error Messages, and Wording - may well contribute to the positive factors that would ‘ultimately determine award-winning sites’ (Sinha et al., 2001). See further discussion in Chattratichart & Brodie (2002b).

5.4.4 HE-Plus: Study 2

This study was carried out by the author of this thesis while being employed at London Metropolitan University. For this reason, only relevant materials are presented here. The purpose to refer to this study is to provide empirical evidence that HE-Plus is easier to use than heuristic evaluation. The implication of this is that our proposed extended framework to the procedure for CDs analysis may indeed be easier than the original procedure.

Hypothesis

Two hypotheses were formed based on the findings in Study 1 as follows:
Hypothesis 1. Evaluators would find HE-Plus easier than heuristic evaluation.
Hypothesis 2. HE-Plus would outperform heuristic evaluation as found in Study 1.

Method

There were two experiments (Group 1 and Group 2) of the same design but for two shopping centre web sites. In Group 1, ten MSc students at CCTM Department (Computing, Communications Technology and Mathematics), London Metropolitan University, evaluated Meadow Hall shopping centre site (http://www.meadowhall.co.uk/home.cfm). In Group 2, nine MSc students from the same institution evaluated Merry Hill shopping centre site (http://www.merryhill.co.uk/home.html). One participant in Group 1 was later found to be an expert usability engineer so her data were not included in this study. The numbers of evaluators in HE and HE-Plus groups were then equal for both Group 1 and Group 2. The experimental procedure, the design and data analysis for both experiments were similar to that of Study 1. Therefore only the differences are described below.

Participants in this study had various prior experience in heuristic evaluation. Therefore, they were asked to rate their own expertise in doing heuristic evaluation on a scale of 1 (novice) to 5 (expert) in the pre-test questionnaire. This information was later incorporated into the new reliability metric used in this study.

Results from Study 1 suggested that the HE-Plus procedures used originally were too rigid and needed to be refined. Hence, in this study, the HE-Plus procedure given in the training packs was simplified. Evaluators were told to be aware of the common problem areas that they had to look out for without being given steps to follow strictly as in Study 1. In addition, a crib sheet was placed next to computers while evaluators did the evaluation as a reminder. The sheet contained the list of the ten heuristics as used in Study 1 for those who did heuristic evaluation; while for those doing HE-Plus, the sheet contained both a list of heuristics and of problem areas found in Study 1 as listed in Table 5.14.

The two experimental sessions for Group 1 and Group 2 took place at the same time in two different computer laboratories and lasted for one and a half hour. When the sessions finished, evaluators submitted their reports on a floppy disk and completed a post-hoc questionnaire about their experience with the web sites, the methods used, and the confidence in their own evaluation. This questionnaire was identical to the one used in Study 1.

Results

Reliability Metric

Evaluators' expertise in this study varied remarkably, depending on their subject area of study. Those who reported higher expertise tended to find more problems than those with lower expertise. For fair comparison, a new metric (OLP) that is also a function of evaluators' expertise was devised to measure overlap between evaluators' findings.

$$\text{OLP} = \frac{\text{Total number of evaluators who find the same problem}}{(\text{Total number of unique problems}) \times (\text{Average group expertise})}$$

Findings

Average ratings, on a scale of 1 to 5 (Table 5.15) revealed that evaluators found HE-Plus easier to use than the original heuristic evaluation method and that they were more confident in their own evaluations than the HE group. In terms of reliability, Kolmogorov-Smirnov test revealed a significant difference of OLP between HE and HE-Plus for Meadow Hall site only, $Z = 1.703$, $p < 0.01$. The problem areas found by HE-Plus evaluators were the same as the ones (derived from Study 1) originally given to them.

Table 5.15 Ratings of the methods

	Group 1 Meadow Hall			Group 2 Merry Hill		
	HE	HE-Plus	User testing	HE	HE-Plus	User testing
Usability of web site	2.7	3.3	3.2	2.9	2.5	2.9
Usability of the method used	2.8	4.0	n/a	3.4	3.6	n/a
Confidence in own evaluation	3.3	4.0	n/a	3.8	4.0	n/a

Note that user testing data come from another study that was also carried out in the same occasion.

5.4.5 General discussion for He-Plus studies

Both HE-Plus: Study 1 and HE-Plus: Study 2 gave positive indications as to the usefulness of HE-Plus for web site evaluation. Study 1 found a distinct superiority in performance (i.e. the overlapping of results) of HE-Plus evaluators over those using the original heuristic evaluation method. Results from Study 2 also showed HE-Plus performed significantly better than heuristic evaluation for Meadow Hall site, although not with the Merry Hill site. Nevertheless, participants' opinion about the new method has improved in the second study. The same was true with the evaluators' confidence in their own results. This indicates that we are heading in the right direction to simplify the procedure for HE-Plus. The lessons learned from this and future refinement of the HE-Plus procedure might well turn to be useful to help structure the procedure of the original heuristic evaluation method so that its reliability can be improved.

We tabulate the data from Study 1 and Study 2 against those of others employing laboratory testing (so-called 'user testing') in the literature (Kessner *et al*, 2001 and Molich *et al*., 1999) in Table 5.16. We can see from these data that, for both user testing and predictive evaluation (inspection) methods, the more focused the evaluation is, the better overlapping of results can be obtained. The results from HE-Plus studies show that the problem areas in the *'Usability Problems Profile'* provide sensitising concepts for evaluators, especially novices, as to where to look out for problems and thus helps evaluators to be more focused. Therefore, Chattratichart *et al*. (2003) suggested that another area of application of the HE-Plus method is in training students and novice usability engineers to do heuristic evaluation.

Table 5.16 Results from different comparative evaluation studies

	Study 2				Study 1[a]		Kessner[b]	Molich[c]
Evaluators	MSc students				Research students		Usability practitioners	
Product evaluated	Meadow Hall shopping centre		Merry Hill shopping centre		Lakeside shopping centre		Dialog box prototype	Hotmail. com
Evaluation method	HE	HE-Plus	HE	HE-Plus	HE	HE-Plus	User testing	User testing
Requests to evaluators	Open-ended	Open-ended	Open-ended	Open-ended	Open-ended	Open-ended	6	29
No. of evaluators finding a problem	1.39	1.53	1.44	1.41	1.19	2.06	2.14	1.32
OLP	0.70	1.02	0.66	0.57	n/a	n/a	n/a	n/a
Usability of the method	2.8	4.0	3.4	3.6	4.6	3.1	n/a	n/a
Confidence in own results	3.3	4.0	3.8	4.0	4.8	4.1	n/a	n/a

a. - from Chattratichart & Brodie (2002); b and c - from Kessner et al (2001).

In the context of this PhD thesis, this section on the HE-Plus method demonstrates how the outcomes from the Prograph study can be applied in other research context. Firstly, if a '*Usability Problems Profile*' is unknown, it may be worthwhile at the early phase of an application development to carry out a detailed usability study as we have done here with Prograph to obtain a *'Usability Problems Profile'* and/or a CDs profile to be used as proposed for the extended framework to CDs. Secondly, this process of arriving at an extended method is not necessarily limited to CDs analysis. It can be used with other inspection methods as well. We have already shown that it could be applied to heuristic evaluation, which is a predictive evaluation method like CDs analysis. Thirdly, pointed out above, overlapping of results was improved in user testing as well as heuristic evaluation when evaluators were kept focused. We therefore suggest that *'Usability Problems Profile'* could also be incorporated into the design of user testing to help improve its results.

5.5 Chapter summary

The focus of this chapter is on usability evaluation of VPLs in search for a suitable evaluation method to be used during the iterative design life cycle of a new VPL. From the review of existing research it can be concluded that the Cognitive Dimensions of Notations (CDs) is the most suitable evaluation technique for evaluating usability of a programming language. Its advantage is cost-effectiveness. Unlike other usability inspection methods, CDs analysis can be quick and cheap to be carried out while also giving a wide and deep coverage of the product being evaluated. However, it has some weaknesses, most seriously, being the reliability of its evaluation results. This, we argued, is due to the vast analysis space that evaluators have, which renders ad hoc evaluation results. Therefore, it is suggested that the reliability could be improved by reducing the analysis space for the evaluators. To do that, we needed to know potential usability problem areas associated with the programming languages of interest. This subsequently formed the research questions for the diary study presented in this chapter.

Results of the diary study revealed ten usability problem areas with varying severity ratings. Pareto analyses were conducted, based on usability problem areas and the dimensions in CDs. The analyses provide a subset of the dimensions of CDs to form an empirically justified CDs profile and a '*Usability Problems Profile*'. The former has been defined by Britton, & Kutar (2001) as 'the desirability of each dimension for a specific activity', which is traditionally derived analytically. The latter, however, is defined by us to refer to typical usability problem areas of concern found in the same type of products. From these profiles, we proposed an extended framework for the original CDs analysis to include an additional contextualised layer of '*Usability Problems Profile*' into its procedure. We envisage that the '*Usability Problems Profile*' will keep evaluators focused and hence the evaluation results can be more reliable. Indeed, there is empirical evidence that confirm the transferability of this finding. Two empirical studies (three experiments) that provide such evidence were subsequently presented. Finally, we discussed how the outcomes of the Prograph study could be applied to other research contexts as demonstrated by these two studies.

6. SYNTHESIS: A PROPOSED SET OF VPL PRINCIPLES AND THEIR EVALUATION

6.1 Introduction

The work presented in this thesis covers a few different areas. The outcomes, resulting from critiques and analyses of previous research in the literature, are a Visual Language Matrix (VLM) for visual programs and six principles for making diagrammatic notations 'good programming languages' (Fitter & Green, 1979). Five controlled experiments (Study units 1 to 5) presented in Chapters 3 and 4 provide answers to a few narrowly focused research questions pertaining to programming paradigms, directional representation and representation of traversal direction (the direction in which a representation is most easily traversed from start to ending). The Prograph study presented in Chapter 5 (Study unit 6) resulted in a list of usability problems found in Prograph. The chapter also discussed and proposed an evaluation framework to be used in evaluating a VPL during its design life cycle. In order to further enhance the contribution of these individual findings, this chapter provides synthesis of practical recommendations.

It is envisaged that, except for our targeted novice users, two other parties who directly benefit from the work of this research are VPL designers/developers and the usability experts of a design team. Whilst the usability experts of the design team can benefit from the evaluation framework proposed in Chapter 5, the designers/developers would benefit more from design principles, guidelines, or checklists. Even though the six principles for designing diagrammatic notations summarised in Chapter 2 can provide some guidance to designers and developers, they are broad, non-contextual, and hence difficult to put into practice. These principles can be made *operational*, however, if they are given in the form of a checklist to help remind designers/developers of important issues to consider during design.

This chapter shows how the work presented and the data obtained earlier in this research is analysed to form the checklist, which is empirically grounded, summarises a set of design principles from the checklist, and triangulates them with findings from other research. The process of obtaining these principles and their evaluation is depicted in Figure 6.1.

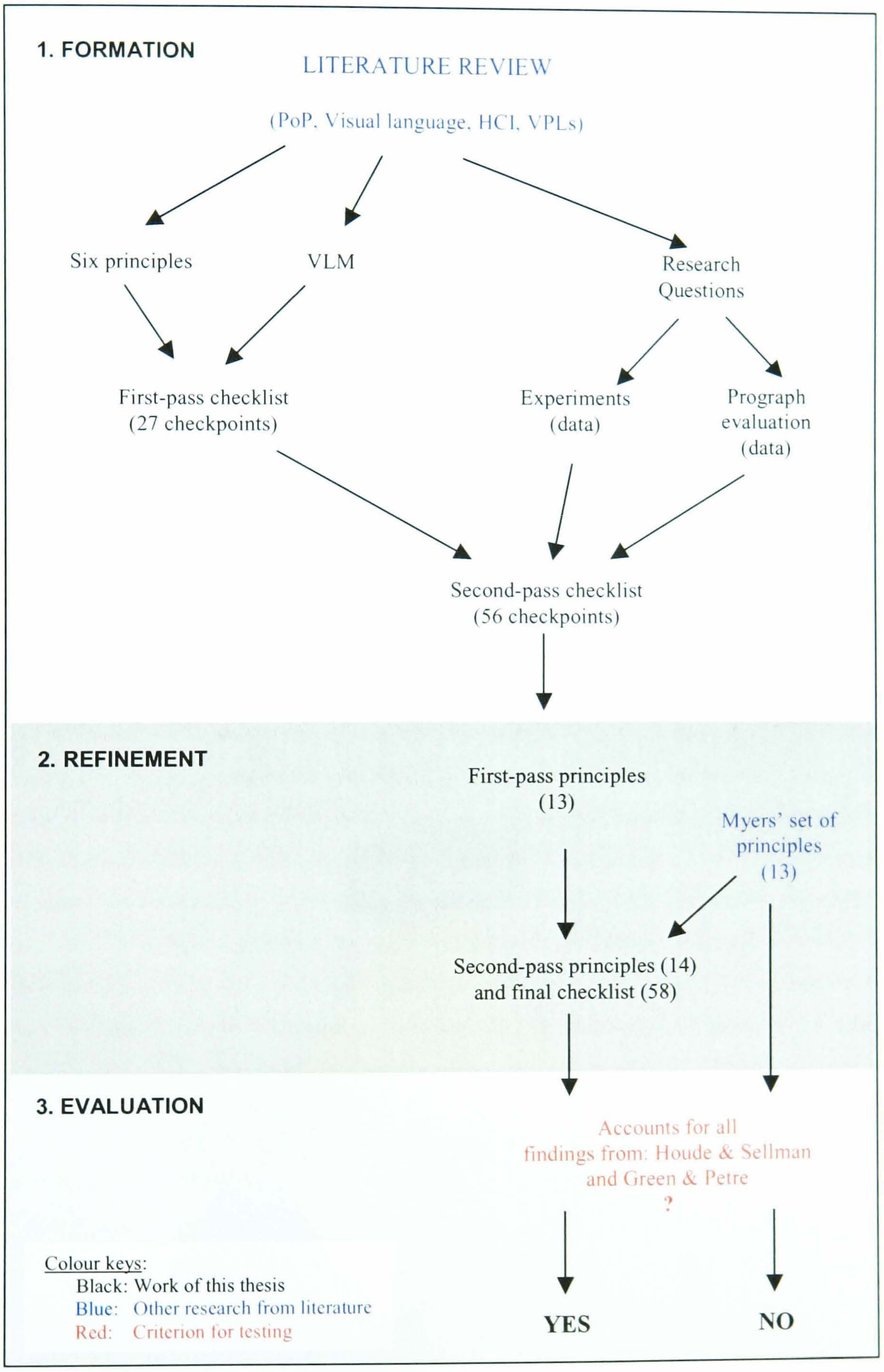

1. FORMATION
LITERATURE REVIEW
(PoP, Visual language, HCI, VPLs)
Six principles
VLM
Research Questions
First-pass checklist (27 checkpoints)
Experiments (data)
Prograph evaluation (data)
Second-pass checklist (56 checkpoints)
2. REFINEMENT
First-pass principles (13)
Myers' set of principles (13)
Second-pass principles (14) and final checklist (58)
3. EVALUATION
Accounts for all findings from: Houde & Sellman and Green & Petre ?
YES
NO
Colour keys:
Black: Work of this thesis
Blue: Other research from literature
Red: Criterion for testing

Figure 6.1 Process of forming-refining-and-evaluating of VPL principles

6.2 Formation: generating checklists

First and foremost, it must be emphasised here that the checklist is to be used as a quick and cheap tool by non-usability experts, i.e. the designers/developers in a design team, as a guide to designing elements or activities that might help improve usability of the VPL being designed. In the first attempt, a 'to-do and not-to-do' list was generated based upon previous research in the literature and the findings of Study units 1 to 6 in this thesis. At this point, no attempt to justify it by conducting yet another naturalistic inquiry or more laboratory experiments for any hypothesis that might be formed will be made.

The procedure for obtaining the checklist (as depicted by Figure 6.2) is carried out as follows. Firstly the 38 design elements in the VLM and the six design principles in Chapter 2 are considered together to generate the first-pass checklist. This checklist is subsequently augmented by the empirical data from Chapter 3, Chapter 4, and the scripts of the diary obtained during the evaluation of Prograph (the Diary Study) in Chapter 5. This yields the second-pass checklist that is refined further later in the refinement phase in Section 6.3. A more detailed procedure for generating the second-pass checklist is described in Section 6.2.2.

The design elements in the VLM and the six principles mentioned above are listed in Appendices D-1 and D-2. The first-pass and second-pass checklists consist of 27 and 56 checkpoints, respectively. They can be found in Appendices D-3 and D-4, respectively.

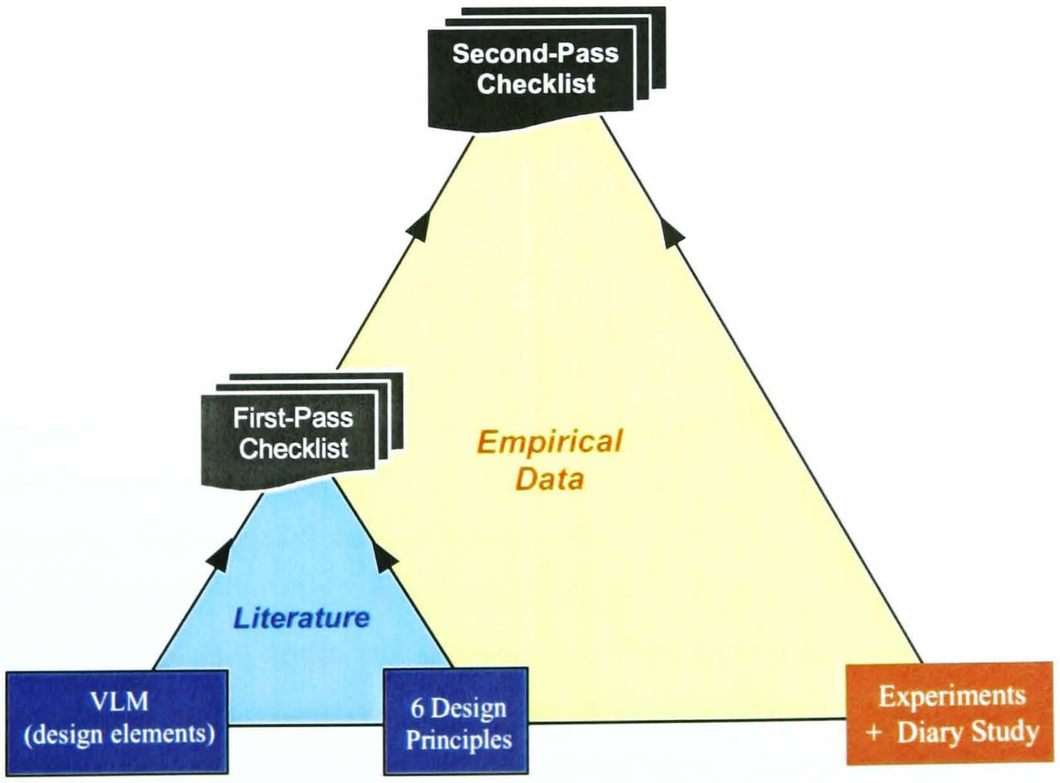
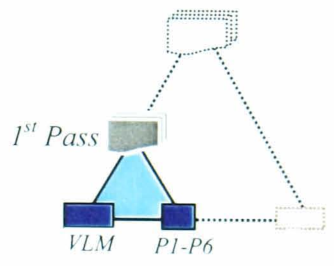


Figure 6.2 Formation phase

6.2.1 First-pass checklist

Twenty-seven checkpoints for the first-pass checklist are generated by the procedure described as follows:

The six design principles, P1 to P6 in Appendix D-1, together with the 38 design elements in the VLM for visual programs listed in Appendix D-2, are used. These design elements are grouped by design modes: text, spatial, and graphics. The miniature representation to the right highlights the part of Figure 6.2 (formation phase) that is now being focused upon, i.e. the generation of first-pass checklist. Twenty-seven checkpoints for the first-pass checklist are generated by this procedure. These are checkpoints T1 to T10; SP1 to SP5; G1 to G11; and GEN-1 listed in Appendix D-3. Checkpoints prefixed with a T, SP, and G refer to text, spatial, graphic modes, respectively. Checkpoints prefixed with GEN refer to 'general' guidance not specific to any of the three modes. The following explains how each checkpoint in the first-pass checklist is obtained.



It must be noted here that the checkpoints generated from this procedure do not form a complete list, that they are the results of our structured generation process, and that they are still subject to further refinement at a later stage in this chapter.

P1 Provide appropriate means and level of abstraction

It is difficult to know what 'appropriate' means without some kind of measurement scheme. Justifying a scheme is also difficult without testing it with users. However, abstraction implies encapsulation of a segment of program code that makes up a function or a module. The more modules or functions there are, the higher abstraction is. In a visual program, a function or module is usually represented by a node or an icon, which must be clicked opened into a new window. The higher level the abstraction is, the more functions or modules a visual program has, and hence the more windows would be opened during programming. These windows contain related code. It is therefore likely that they will be left visible on the screen until the particular task (using the section of the program code) is finished. A high level of abstraction can therefore be associated with many windows, each of which consists of only a few programming objects. On the other hand, a low level of abstraction can be associated non-modular programming style and hence, with a few messy looking windows (bad layout or long scrollable length in each window).

Two checkpoints (G5 and G6 in Appendix D-3) are generated for the checklist:

G5:	Avoid too much abstraction (Do you see too many windows opened or just a few objects per window?).
G6:	Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Is scrolling required excessively?).

P2 Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner

Symbols, graphical objects and names should be made visible, easily discriminated, role expressive and not error-prone. Standards and conventions should be followed as far as possible. Language use should also clearly convey the intended meanings. These contribute to many checkpoints in the first-pass checklist:

T1:	Use appropriate font size.
T3:	Use lower case or sentence case.
T4:	Use trigger words, meaningful names or symbols.
T5:	Use easy language for dialogues, help, text and error messages.
T7:	If colour-coding is used, use colours that stand out.
T9:	Use numbers or letters as points of reference across screens/views.
T10:	Use standard symbols and operators (for example, using γ or <> for 'not equal').
SP1:	Make sure that object size is not too small to be noticeable on a messy screen.
SP2:	Avoid messy windows. Neat layouts should be achieved easily and quickly.
SP4:	Keep the position of the same object consistent in different windows as far as possible.
G1:	Use familiar icons and those that match real world objects, e.g. ✓ for ticks, × or x for crosses.
G2:	Use familiar or standard representations for programming constructs (such as a diamond shape for decision point as in flowcharts).
G4:	Exploit standard conventions (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).

G7:	Use colour coding and shading (as a second means to convey a meaning). Use them in a consistent manner.
G8:	Make icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.
G10:	Use two-dimensional representations as far as possible. If three-dimensional ones must be used, use them effectively.

P3 Use secondary notation as appropriate

Secondary notation provides a second means to convey a meaning but is not part of the notation associated with the language (Green & Petre, 1996). Indentation in textual programs and coding by colours and by object shapes are examples of secondary notation. Careful use of secondary notation can promote ease of understanding. This is achieved by, for example, selecting colours and shapes according to standard and convention, avoiding overloading the short-term memory by using too many different shapes or colours, and so on. Tradeoffs must be carefully considered.

These constitute checkpoints T6, G3 and G7 in Appendix D-3 below:

T6:	May use colour-coding in labels, names of different categories, types or groups.
G3:	Implement coding-by-shape.
G7:	Use colour coding and shading as a second means to convey a meaning. (Use them in a consistent manner).

P4 Support modification through simplicity, clarity, and flexibility

Changes to the program should not be too difficult. Modification can be supported by providing a tool, which the user can use to sketch a program quickly and which the user does not have to commit to a full program before executing it, i.e. the system being flexible. Making representations, names and labels clear and simple also helps support modification. Dependency between entities in the notation should be made explicit by having some form of links or references such as linework, numbers or alphabets.

In light of this principle, five checkpoints are generated for the checklist as follows:

T2:	Stay simple with fonts – do not use fancy and different font types.
T9:	Use numbers or letters as points of reference across windows/views.
SP4:	Keep the position of the same object consistent in different windows as far as possible.
G11:	Provide some kind of tree-structure to make referencing visible.
GEN-1:	Provide a low fidelity tool (for example, provide functionality that supports quick and easy modification of the program code but yet does not require precision).

P5 Support evaluation

Opportunistic design can be supported through animation or making partial code executable. These are listed as G9 and GEN-1 in Appendix D-3:

G9:	Provide animation where appropriate such as in debugging tools.
GEN-1:	Provide a low fidelity tool (for example, provide functionality that allows programmers to quickly assess or test a program even though the program is incomplete).

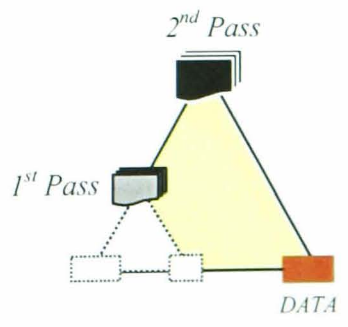
P6 Offload cognitive efforts required where possible

This can be achieved through checkpoints T8, SP3, SP5 in Appendix D-3:

T8:	Do not using too many colours in the colour-coding scheme.
SP3:	Avoid complex traversing rules.
SP5:	Avoid scrolling or keep it to minimum.

6.2.2 Second-pass checklist

In this section, we describe the procedure that generates the second-pass checklist, which can be found in Appendix D-4. The miniature representation on the right highlights this stage of the formation phase (see Figure 6.2).



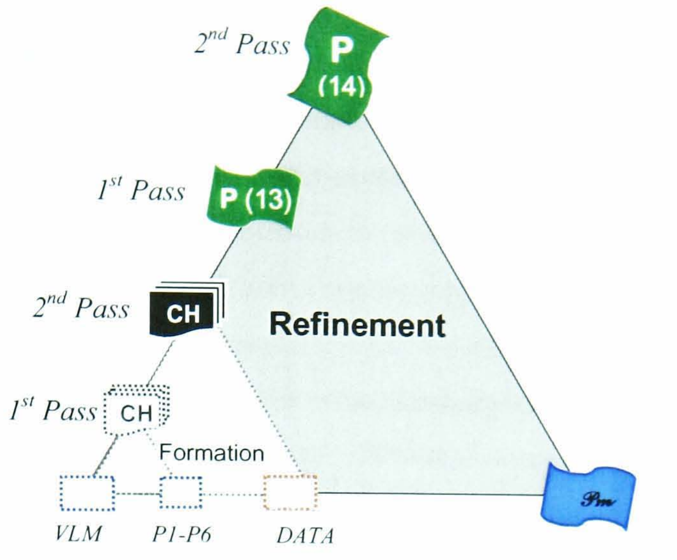
The second-pass checklist is generated by matching each problem found during the Prograph evaluation (Appendix C-2) and the findings from our experiments in Chapters 3 and 4 with the checkpoints in the first-pass checklist (Appendix D-3). Where there is no match, a new checkpoint description reflecting the problem token concerned is added into the checklist. An example of this is checkpoint T12 in Appendix D-4, "allow users to edit a default name". This is not in the first-pass checklist and is generated as a result of this process. This process is repeated for all the problems in Appendix C-2, resulting in 58 checkpoints as a potential second-pass list in Appendix D-4. All but two checkpoints in the first-pass checklist match one or more problems found in the Prograph evaluation (Appendix C-2). The statistics for these two checklists (first-pass and second-pass) are presented in Table 6.1. The two unmatched checkpoints generated in the first-pass are SP4 ("keep the position of the same object consistent in different windows as far as possible" and G3 ("implement coding-by-shape"). Therefore they are removed from the potential second-pass checklist, yielding the second-pass checklist that consists of 56 checkpoints in total.

Table 6.1 Statistics of the two checklists

	Number of checkpoints
First-pass checklist	27
Second-pass checklist	56
In first-pass and matches empirical data	25
In first-pass but does not match empirical data	2
New additions to first-pass	31

6.3 Refinement

This section describes the process used to refine the second-pass checklist to first-pass and second-pass principles. Myers (n.d.) suggests 13 principles for good textual programming languages based on the ten heuristics used for Nielsen's (1993) Heuristic Evaluation. These 13 principles (Myers, n.d.) were used to refine our first-pass principles. Figure 6.3 depicts the refinement phase of the synthesis in this chapter.

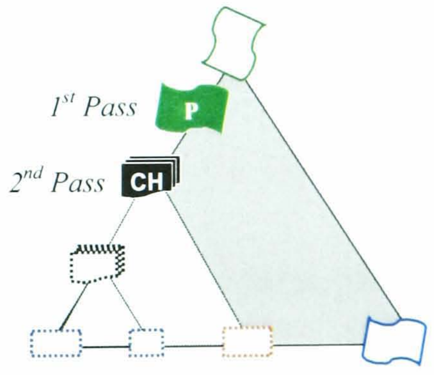


(Note: CH = checklist; P = Principles;Pm = Myers' (n.d.) principles)

Figure 6.3 Refinement phase

6.3.1 First-pass principles

In the second-pass checklist, 56 checkpoints have been generated from six design principles and the design elements in the VLM summarised in Chapter 2, augmented with empirical data from this research. This section takes a reverse approach in an attempt, with the data, to either agree or disagree with the six principles, with a view to forming a new set of empirically grounded principles for diagrammatic VPLs. This is done (as shown in Appendix D-5) by assigning the keys to the original six design principle(s) given in Appendix D-1 to each checkpoint in the second-pass checklist generated in Appendix D-4. Where none of the keys in Appendix D-1 is appropriate, a new key is assigned and its description is entered in Appendix D-6, which has been half-filled with the original six principles P1 to P6 in Appendix D-1. Similarly, this procedure is carried out with VLM design elements M1 to M38 in Appendix D-2, resulting in only one new key to be added into the VLM – "Components within a graphical object". The miniature representation on the right highlights this stage (of generating the first-pass principles) in Figure 6.3.



(Note: P = Principles; CH = checklist)

The new set of design principles, or first-pass principles as listed in Appendix D-6 consists of the following:

First-pass principles

P1	Provide appropriate means and level of abstraction.
P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
P3	Use secondary notation as appropriate.
P4	Support modification through simplicity, clarity, and flexibility.
P5	Support evaluation.
P6	Offload cognitive efforts required where possible.
P7	Support minimalism and economy of interaction.
P8	Operation on devices should meet user's expectation.
P9	Encourage user's control and freedom.
P10	Avoid hard concepts.
P11	Beware of misleading appearance.
P12	Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension.
P13	Ensure consistency in provisions (e.g. of functions) and their implementation

Checkpoints for the second-pass checklist are organised by the first-pass principles as follows:

P1 *Provide appropriate means and level of abstraction*

1. Avoid too much abstraction – Do you see: a) too many windows opened or b) just a few objects per window?
2. Avoid too little abstraction – Do you see objects dispersed everywhere in the same window which could have been grouped? Is excessive scrolling required?

P2 *Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner*

Discriminability

1. Use a comfortable font size.
2. If colour-coding is used, use the colours that stand out.
3. Ensure that multiple floating windows/views of code are distinguishable from one another by visible and noticeable difference in titles.
4. Use a comma to separate items in a horizontal list rather than a space.
5. Make sure that object size is not too small to be noticeable so users do not have to search for it.
6. Allow adequate separation between different parts of a graphical primitive.

7. Make the icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.
8. Use two-dimensional representations as much as possible. If three-dimensional representations must be used, use them effectively.
9. Make windows/views distinguishable from one another by making use of visible and noticeably different icons.
10. Use lower case or sentence case.

Familiarity

11. Use standard symbols and operators (for example, using γ or <> for 'not equal').
12. Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).
13. Use familiar icons and those that match real world objects, e.g. ✓ for ticks, ✗ or x for crosses.
14. Use familiar or standard representations for programming constructs and functions.
15. Make manipulation of objects (e.g. resizing) intuitive in both directions for paired operations, e.g., copy/delete, shrink/enlarge.

Meaning/language

16. Use trigger words, meaningful names or symbols.
17. Use easy language for dialogues, help, text and error messages.
18. Use consistent naming convention (upper/lower case, brackets, quotation marks, etc.).
19. Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity.

Layout

20. Avoid messy windows. Neat layouts should be achieved easily and quickly.
21. The most current window/view must not cover the one leading to it. They are better side-by-side.
22. Provide a facility to tidy up and straighten links.

Reference

23. Use numbers or letters as points of reference across windows/views.
24. Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program.

P3 *Use secondary notation as appropriate*

1. Use colour-coding and shading as a secondary means to convey a meaning. Use them in a consistent manner.
2. May use colour-coding in labels, names of different categories, types, or groups.

P4 *Support modification through simplicity, clarity, and flexibility*

1. Provide a low fidelity tool.
2. Stay simple with fonts – do not use fancy and different font types.
3. Allow users to edit a default name.

P5 *Support evaluation*

1. Provide animation where appropriate such as in debugging tools.

P6 *Offload cognitive efforts required where possible*

1. Avoid scrolling or keep it to minimum.
2. Do not use too many colours in the colour-coding scheme for textual messages, titles and names.
3. Avoid complex traversing rules.

P7 *Support minimalism and economy of interaction*

1. Remember that too much automation is not good sometimes.
2. Provide an icon for quickly starting a new task such as a new project. That is, make the first initial step easy to figure out.
3. Provide icons for some frequently used functions for easy access (undo, execute).
4. Provide undo ability for all operations in manipulating objects (delete, copy, grouping).
5. Automatically adjust the object to an appropriate size.
6. Assign only one primitive to include a few operations that are frequently used together to do a task.
7. Allow code to be created on the fly – any time; even while the program is running.

P8 *Operation on devices should meet user's expectation*

1. Make appropriate use of left and right mouse-clicks for different tasks or functions on the same object (as would be expected by users). Otherwise, it only causes confusion.

P9 ***Encourage user's control and freedom***

1. Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order for doing things (in creating objects and links, defining attributes, etc.).
2. Users can add comments at any time and anywhere and are free to hide or show the comment made.

P10 ***Avoid hard concepts***

1. Implement an easy way to pass controls. Do not be too restrictive.
2. Make loop termination simple and require no thinking ahead.
3. Make Iteration easy.
4. Make method referencing easy.

P11 ***Beware of misleading appearance***

1. Avoid misleading users by having a part in the object that looks meaningful but is meaningless or never used.

P12 ***Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension***

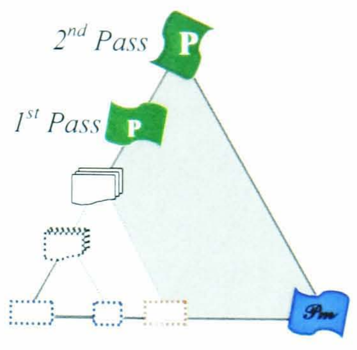
1. Use graphics in the HELP document – make it visual.
2. Ensure Help provides a full coverage of all operations and functions.
3. Provide a list of the exact names of operators or fuzzy search facilities.
4. Ensure Help does not provide incorrect or outdated information.
5. Provide adequate information in error messages.
6. A previous error message should either disappear or make it known that it is not applicable now.
7. Use easy language for dialogues, help, text and error messages.

P13 ***Ensure consistency in provisions (e.g. of functions) and their implementation***

1. Do not provide a feature or function that is not meant to be available.
2. Make all available features work.
3. Debug the application thoroughly.

6.3.2 Second-pass principles

This section describes the process used to refine the first-pass list to obtaining 14 principles in the second-pass list (as highlighted in the miniature representation on the right). The principles for programming languages (textual in general) given by Myers (n.d.) are considered in this process. This results in the second-pass principles, which consists of 14 principles and 58 checkpoints in all.



(Note: P = Principles; Pm = Myers' principles)

The following are Myers' (n.d.) principles for good programming languages based on Nielsen's (1993) heuristics:

1. Good graphic design and colour choice.
2. Less is more ("keep it simple").
3. Speak the user's language.
4. Use appropriate mappings and metaphors.
5. Minimise user memory load.
6. Be consistent.
7. Provide appropriate feedback.
8. Clearly marked exits.
9. Prevent errors.
10. Good error messages.
11. Provide shortcuts.
12. Minimise modes.
13. Help the user get started with the system.

For each of his principles, Myers (n.d.) also gives example problems. We consider each of his examples as to which principle(s) in the first-pass principles in Section 6.3.1 it corresponds to. This information is tabulated in Table 6.2, which also shows the list of second-pass principles generated from this matching process. Not all the first-pass principles account for Myers' example problems and principles. The two bottom rows of Table 6.2 are cases where Myers' examples did not correspond to any of the first-pass principles. They are thus assigned as additional principles in the second-pass principles. Myers' Principle 9 ("Prevent errors"), which is located in the last row of Table 6.2, has no direct match with any of the principles in the first-pass list so it is assigned as Principle 14 for the second-pass list. Principle 11 of the first-pass list, "Beware of misleading appearance", does not match have a direct match with Myers' examples. However, this principle implies prevention of error so it is included in Principle 14 of the second-pass list. All cells in Table 6.2 that contain discrepancies (mismatch between items in our first-pass list and Myers' examples are shaded in grey. The new revised set of principles, the second-pass principles, consists of 14 principles. The description of Principle 2 has been changed slightly because empirical data indicate the relevance. The final set in the second-pass principles are given as follows:

Second-pass principles

P1	Provide appropriate means and level of abstraction.
P2	Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner.
P3	Use secondary notation as appropriate.
P4	Support modification through simplicity, clarity, and flexibility.
P5	Support evaluation (by providing suitable functionality).
P6	Offload cognitive efforts required where possible.
P7	Support minimalism and economy of interaction.
P8	Operation on devices should meet user's expectation.
P9	Encourage user's control and freedom.
P10	Avoid hard concepts.
P11	Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension.
P12	Ensure consistency in provisions (e.g. functions) and their implementation.
P13	Ensure consistency in the ways things are done.
P14	Prevents or corrects for errors (by providing appropriate automated functionality and by avoiding misleading appearance).

Table 6.2 Generating second-pass principles

Second-pass principles[1]	First-pass principles	Description	Myers' (n.d.) Principles		
			Principle	Description	Examples cited
P1	P1	Provide appropriate means and level of abstraction.	None		
P2	P2	Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner.	1	Good graphic design and colour choice	Colour-coding comments and errors as in Visual Basic.
			3	Speak the users language	Use familiar language and symbols in programming syntax.
			4	Use appropriate mappings and metaphors	Syntax agrees with knowledge from other domain (assignment statement in programming conflicts with that in mathematics).
			6	Be consistent	Be consistent in the use of symbols, punctuations.
			8	Clearly marked exits	Make 'exit' from loop or function clear.
P3	P3	Use secondary notation as appropriate.	1	Good graphic design and colour choice	Make good use of layout and indentation in writing programs.
P4	P4	Support modification through simplicity, clarity, and flexibility.	None		
P5	P5	Support evaluation (by providing suitable functionality).	None		
P6	P6	Offload cognitive efforts required where possible.	5	Minimise user memory load	Don't have too many syntax/special rules. Don't make programmers have to remember all the functions and their parameters.

[1] *Generated from the matching process utilising data from the other columns.*

Table 6.2 (continued)

Second-pass principles[1]	First-pass principles	Description	Myers' (n.d.) Principles		
			Principle	Description	Examples cited
P7	P7	Support minimalism and economy of interaction.	12	Minimise modes	Compile versus run mode.
			13	Help the user get started with the system	Program size is much too large for what it does. Be concise.
			2	Less is more ("keep it simple")	Have only a few different features and small number of basic concepts.
			9	Prevent errors	Don't have too many steps in doing things, etc.
			11	Provide shortcuts	
P8	P8	Operation on devices should meet user's expectation.	None		
P9	P9	Encourage user's control and freedom.	None		
P10	P10	Avoid hard concepts.	None		
	P11	Beware of misleading appearance.	None		
P11	P12	Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension.	10	Good error messages	Give sufficient and relevant information.
			7	Provide appropriate feedback	Give feedback more often. Don't wait until compilation or run time.
P12	P13	Ensure consistency in provisions (e.g. of functions) and their implementation	6	Be consistent	Be consistent in the provision of automation.

[1] *Generated from the matching process utilising data from the other columns.*

Table 6.2 (continued)

Second-pass principles[1]	First-pass principles	Description	Myers' (n.d.) Principles		
			Principle	Description	Examples cited
P13	?	Ensure consistency in the ways things are done.	6	Be consistent	Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules –make it applicable in all situations, etc.
P14	?	Prevents or corrects for errors (by providing appropriate automated functionality and by avoiding misleading appearance).	9	Prevent errors	Provide error-checking facility.
					Provide automatic garbage collector to prevent errors of memory management.
					Provide automatic spell-check facility.

[1] *Generated from the matching process utilising data from the other columns.*

<u>A note on the changes from the first-pass principles to the second-pass principles</u>

The two lists: first-pass and second-pass are in the same order from 1 to 10. From P11 onwards, they are different. For clarification, P11 to P14 of the second-pass list are given below along with their checkpoints.

P11 Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension

1. Use graphics in the HELP document – make it visual.
2. Ensure Help provides a full coverage of all operations and functions.
3. Provide a list of the exact names of operators or fuzzy search facilities.
4. Ensure Help does not provide incorrect or outdated information
5. Provide adequate information in error messages.
6. A previous error message should either disappear or make it known that it is not applicable now.
7. Use easy language for dialogues, help, text and error messages.

P12 *Ensure consistency in provisions (e.g. of functions) and their implementation*

1. Do not provide a feature or function that is not meant to be available.
2. Make all available features work.
3. Debug the application thoroughly.

P13 *Ensure consistency in the ways things are done*

1. Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules – make it applicable in all situations, etc.

P14 *Prevents or corrects for errors (by providing appropriate automated functionality and by avoiding misleading appearance)*

1. Avoid misleading users by having a part in the object that looks meaningful but is meaningless or never used.
2. Provide automatic facilities such as error-checking, garbage collector, and spell-check facility.

It must be brought into attention that there were two additional checkpoints obtained from the above analysis in Section 6.3.2 that must be added into the second-pass checklist, which consists of 56 checkpoints, in Appendix D-4. These are:

Checkpoint 57:

Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules – make it applicable in all situations, etc.

Checkpoint 58:

Provide automatic facilities such as error-checking, garbage collector, and spell-check facility.

The above two checkpoints are then added into the second-pass checklist, yielding 58 checkpoints in total. They are listed in Appendix D-8 and called 'Refined second-pass checklist'.

6.4 Evaluation

The work of Houde & Sellman (1994) and of Green & Petre (1996) have already been discussed in Chapter 5, Section 5.3.3 and therefore will not be described here again. In this section, we triangulate the second-pass principles and Myers' (n.d.) principles against findings of usability evaluation research by Houde & Sellman (1994) and by Green & Petre (1996). We match each problem found in their research (Houde & Sellman, 1994; Green & Petre, 1996) with the principles in the second-pass principles (also provided in Appendix D-7) and in Myers' (n.d.) set of principles in Section 6.3.2.

6.4.1 Analysis of the problems found by Houde & Sellman (1994)

1. "Standard features such as graphical layout tools, rulers, and alignment commands were missing" (referring to MacPaint and MacDraw).

This research: This problem corresponds to principle, P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 22 in Appendix D-8 —'Use familiar or standard representations for programming constructs and functions'.

Myers' (n.d.): This corresponds to Myers' Principle 4, 'Use appropriate mappings and metaphors' (see Section 6.3.2).

2. "It was not possible to change the original object types..."

This research: This problem corresponds to P4 in Appendix D-7, 'Support modification through simplicity, clarity, and flexibility', checkpoint 12 in Appendix D-8 —'Allow users to edit a default name', but in this context, instead of 'name' it applies to 'type' as well.

Myers' (n.d.): This does not correspond to any of his principles.

3. "...or even to 'copy' the name and position properties of the original fields and 'paste' them into the number fields. This work had to be repeated."

This research: This problem corresponds to P7 in Appendix D-7, 'Support minimalism and economy of interaction', checkpoint 38 in Appendix D-8 – 'Provide icons for some frequently used functions for easy access (undo, execute)'.

Myers' (n.d.): This corresponds to Myers' Principle 11, 'Provide shortcuts'.

4. "He realized that this revision implied changing the library of drawing function included in the project. While making this change, he forgot to update other parts of the program that would be affected and spent several minutes debugging."

This research: This problem corresponds to P4 in Appendix D-7, 'Support modification through simplicity, clarity, and flexibility'. However, none of the checkpoints for P4 (see Section 6.3.1) fits this situation, which indicates a need for some kind of tool that can detect hidden dependency within the program and either reports or makes the anticipated change visible. However, this is taken care of by P14 in the second-pass list – 'Prevents or corrects for errors (by providing appropriate automated functionality and by avoiding misleading appearance)'.
Myers' (n.d.): This corresponds to Myers' Principle 5, 'Minimise user memory load'.

5. "Some referencing problems arose from names which did not evoke the items they represented."

This research: This problem corresponds to P12 in Appendix D-7, 'Ensure consistency in provisions (e.g. of functions) and their implementation', checkpoint 47 in Appendix D-8 – 'Make all available features work'.

Myers' (n.d.): This corresponds to Myers' Principle 6, 'Be consistent' (in this case, it relates to the provision of automation).

6. "The Director programmer...realized that he didn't know which one (of the four fields he created in the cast window) to put where (in the stage window). They all looked the same, and their labels could not be revealed in the stage view."

This research: This problem corresponds to P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', Checkpoint 27 in Appendix D-8 – 'Make the icons/objects look distinctive (distinctly different), use colour, highlights, shading, line weight, and framing to promote discriminability. Make them noticeable'.

Myers' (n.d.): This could perhaps fit into Myers' Principle 1, 'Good graphic design and colour choice' if Myers' definition can be extended beyond the context of colour, layout, and indentation.

7. "He (the HyperCard programmer) would like to simply select all four fields to change all of their text properties at once."

This research: This problem corresponds to P7 in Appendix D-7, 'Support minimalism and economy of interaction', checkpoint 38 in Appendix D-8 – 'Provide icons for some frequently used functions for easy access (undo, execute)'.

Myers' (n.d.): This corresponds to Myers' Principle 11, 'Provide shortcuts'.

8. "Participants could not keep track of all the components required ..."

This research: This problem corresponds to the following:

- P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 30 in Appendix D-8 – 'Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as a tree-structure for method referencing or provide a list of methods created so far in the program'.
- P2 Appendix D-7, checkpoint 9 in Appendix D-8 – 'Use numbers or letters as points of reference across windows/views'.

Myers' (n.d.): This does not correspond to any of his principles.

9. "They forgot where program elements were, what they were called, what state they were in, and what their relationships were to other parts of the program."

This research: This problem corresponds to P6 in Appendix D-7, 'Offload cognitive efforts required where possible but none of the checkpoints matches this problem exactly. The checkpoints 31 and 9, in effect, would help reduce cognitive efforts required.

- Checkpoint 31 in Appendix D-8 – 'Make windows/views distinguishable from one another by making use of visible and noticeably different icons'.
- Checkpoint 9 in Appendix D-8 – 'Use numbers or letters as points of reference across windows/views'.

Myers' (n.d.): This corresponds to Myers' Principle 5, 'Minimise user memory load'.

10. "We noticed that the current state of the program being edited was not effectively represented to users."

This research: This problem corresponds to P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 20 in Appendix D-8 – 'The most current window must not cover the one leading to it. They are better side-by-side'.

Myers' (n.d.): This does not correspond to any of his principles.

11. "The visual identity of the program and its ties to related elements were not clearly represented...It was hard to tell them apart and ..."

This research: This problem corresponds to the following:

- P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 9 in Appendix D-8 – 'Use numbers or letters as points of reference across windows/views'.
- P2, as above, checkpoint 33 in Appendix D-8 – 'Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity'.
- P2, as above, Checkpoint 30 in Appendix D-8 – 'Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program'.

Myers' (n.d.): This does not correspond to any of his principles.

12. "Appropriate views were not always available...the HyperCard programmer had to frequently select graphic elements to bringing up their individual code dialog boxes to review variable names."

This research: This problem corresponds to P1 in Appendix D-7, 'Provide appropriate means and level of abstraction', checkpoint 24 of the final checklist in Appendix D-8 – 'Avoid too much abstraction'.

Myers' (n.d.): This corresponds to Myers' Principle 5, 'Minimise user memory load'.

13. "The serious programmer ...could not access them (the desired views) in the desired order."

This research: This problem corresponds to P9 in Appendix D-7, 'Encourage user's control and freedom'. checkpoint 35 of the final checklist in Appendix D-8 – 'Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order for doing things (in creating objects and links, defining attributes, etc.).

Myers' (n.d.): This does not correspond to any of his principles.

14. "Before editing the graphical layout view, …could not iteratively make changes in both these views easily."

This research: This problem corresponds to P4 in Appendix D-7, 'Support modification through simplicity, clarity, and flexibility', checkpoint 12 in Appendix D-8 – 'Allow users to edit a default name', but in this context, instead of 'name' it applies to 'type' as well. So this must later be incorporated into checkpoint 12 in the Appendix D-8 as 'Allow users to edit a default object properties such as name and data type'.

Myers' (n.d.): This does not correspond to any of his principles.

In summary, Table 6.3 shows that while all 14 problems (100%) reported by Houde & Sellman's (1994) fit into at least one of the principles derived by this research (P1, P2, P4, P6, P7, P9, P12 in the second-pass principles), only 8 problems (57%) fit Myers' (n.d.) principles (Principles 1, 4, 5, 6, and 11).

Table 6.3 Triangulation with Houde & Sellman's (1994) work

Problem	Second-pass principles		Myers' (n.d.) principles
	Checkpoint	Principle	
1	22	P2	4
2	12	P4	None
3	39	P7	11
4	None	P4	5
5	48	P12	6
6	28	P2	1
7	39	P7	11
8	9, 31	P2	None
9	None	P6	5
10	20	P2	None
11	9, 31, 34	P2	None
12	25	P1	5
13	36	P9	None
14	12	P4	None

6.4.2 Analysis of the problems found by Green & Petre (1996)

1. "Methods can be created on the fly" (Chapter 5, Table 5.9).

This research: This corresponds to P7 in Appendix D-7, 'Support minimalism and economy of interaction', checkpoint 42 in Appendix D-8 –' Allow code to be created on the fly'.

Myers' (n.d.): This corresponds to Myers' Principle 11, 'Provide shortcuts'.

2. "List processing is good and makes implementation of loop easy."

The above comment by Green & Petre (1996) is not a problem. According to Green & Petre (1996), Prograph list processing facility was good because the dimension 'Closeness of mapping' – one of the dimensions in CDs (see Chapter 5) was well supported. However, their analysis (Green & Petre, 1996) conflicts with our findings. While they argued (from their analysis using CDs) that list processing was good, the data from our diary showed both advantages and disadvantages of this facility. All problems reported by us on list processing correspond to P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner'.

For the present purpose, i.e. evaluating the second-pass principles, this analysis by Green & Petre (1996) is therefore irrelevant.

3. "Better than textual languages"

This comment by Green & Petre (1996) referred to consistency of VPLs (Prograph and LabVIEW in their study) being better than that of textual languages because VPLs had simpler syntax than textual programming languages.

This research: This corresponds to P13 in Appendix D-7, 'Ensure consistency in the ways things are done'. It is not possible to find a matching checkpoint for this because they did not provide a clear example of 'simpler syntax'.

Myers' (n.d.): This corresponds to Myers' Principle 6, 'Be consistent', described by the following example: 'Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules – make it applicable in all situations'.

4. "Too many windows"

This research: This corresponds to P1 and P2 in Appendix D-7, 'Provide appropriate means and level of abstraction' and 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', respectively. It is consistent with checkpoints 16 and 24 in Appendix D-8 – 'Avoid messy windows…' and 'Avoid too much abstraction', respectively.

Myers' (n.d.): This does not correspond to any of his principles.

5. "Repeated reversals of success and failure controls", referring to control flow constructs implemented in Prograph, this problem violates the dimension, Hard mental operations', of CDs (Green & Petre, 1996).

This research: This corresponds to P10 in Appendix D-7, 'Avoid hard concepts', checkpoint 49 in Appendix D-8 – 'Avoid hard concepts that require thinking ahead in passing control'.

Myers' (n.d.): This corresponds to Myers' Principle 5, 'Minimise user memory load'.

6. "Cannot navigate up the call graph to find which method call which or which is called by which"

This research: This corresponds to P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 30 in Appendix D-8 –'Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as a tree-structure for method referencing or provide a list of methods created so far in the program'.

Myers' (n.d.): This does not correspond to any of his principles.

7. "Commitment to connection, to order of creation"

This research: This corresponds to P9 in Appendix D-7, 'Encourage user's control and freedom', checkpoint 35 in Appendix D-8 –'Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order for doing things…'.

Myers' (n.d.): This does not correspond to any of his principles.

8. "Dummy methods can be created; code can be added or changed at run time"

This research: This corresponds to P7 in Appendix D-7, 'Support minimalism and economy of interaction', Checkpoint 42 in Appendix D-8 –' Allow code to be created on the fly – any time; even while the program is running'.

Myers' (n.d.): This does not correspond to any of his principles.

9. "The tick and cross controls"

This research: This corresponds to P10 in Appendix D-7, 'Avoid hard concepts', Checkpoint 49 in Appendix D-8 –' Avoid hard concepts that require thinking ahead in passing control'.

Myers' (n.d.): This corresponds to Myers' Principle 5, 'Minimise user memory load'.

10. "Diagrams are untidy; cannot use layout to communicate; groups of objects cannot be commented."

This research: This corresponds to the following:

- P1 in Appendix D-7, 'Provide appropriate means and level of abstraction', checkpoint 25 in Appendix D-8 – 'Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped?)'.
- P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', Checkpoint 16 in Appendix D-8 –'Avoid messy windows. Neat layouts should be achieved easily and quickly'.
- P9 in Appendix D-7, 'Encourage user's control and freedom', checkpoint 36 in Appendix D-8 –'Users can comment at any time anywhere and are free to hide or show the comments made'.

Myers' (n.d.): This does not correspond to any of his principles.

11. "Deep subroutine structure" should be made visible.

This research: This corresponds to P2 in Appendix D-7, 'Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner', checkpoint 30 in Appendix D-8 –'Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as a tree-structure for method referencing or provide a list of methods created so far in the program'.

Myers' (n.d.): This does not correspond to any of his principles.

12. "Empirically supported; poorer than Basic"

This is irrelevant because it refers to an experiment comparing the two VPLs (Prograph and LabVIEW) to Basic.

In summary, Table 6.4 shows that while all ten relevant problems (100%) reported by Green & Petre (1996) fit into at least one of the principles derived by this research (P1, P2, P7, P9, P10, and P13 in the second-pass principles). Only four out of ten (40%) relevant problems fit into Myers' (n.d.) principles 5 and 11. This clearly shows that Myers' (n.d.) principles are inadequate for VPLs.

Table 6.4 Triangulation with Green & Petre's (1996) work

Problem	Second -pass principles		Myers' (n.d.) principles
	Checkpoint	Principle	
1	43	P7	11
2	Irrelevant		
3		P13	6
4	25, 16	P1, P2	None
5	50	P10	5
6	31	P2	None
7	36	P9	None
8	43	P7	None
9	50	P10	5
10	26, 16, 37	P1, P2, P9	None
11	31	P2	None
12	Irrelevant		

A comparison of the evaluation results in this and the previous sections is illustrated in Figure 6.4, which shows that the set of VPL principles obtained by this research is superior to that recommended by Myers (n.d.) which is based on mostly textual programming languages and Nielsen's (1993) heuristics.

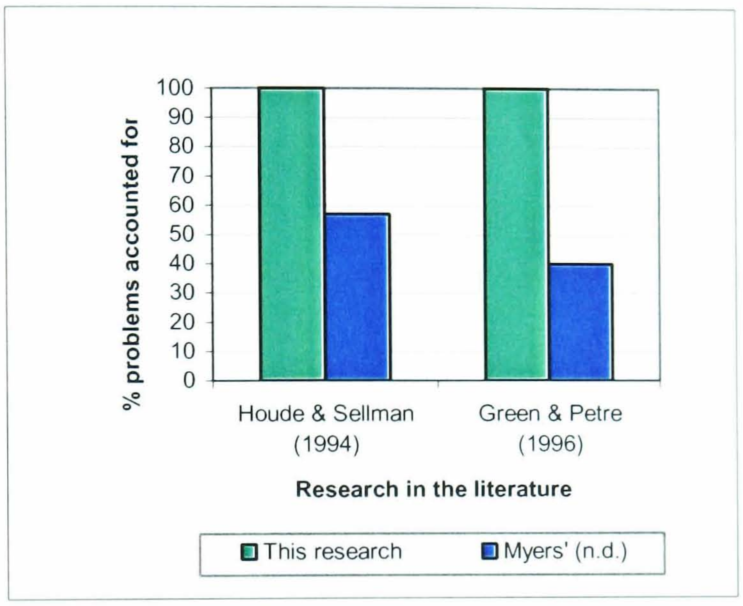


Figure 6.4 **Evaluation of two sets of VPL design principles against the problems reported by two existing studies in the literature**

6.5 Synthesis deliverables: final checklist and principles

This section summarises the final checklist and design principles from the synthesis that has been presented so far. The final checklist consists of 58 checkpoints, which are categorised into 14 principles. They are tabulated in Table 6.5 according to their corresponding principles.

Table 6.5 Final checklist and design principles for diagrammatic VPLs

Design principles and their checkpoints	
Principle 1: Provide appropriate means and level of abstraction	
1	Avoid too much abstraction (Do you see: a) too many windows opened or b) just a few objects per window?).
2	Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Does scrolling in a particular window/view seem endless?).
Principle 2: Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner	
Discriminability	
3	Use comfortable font size.
4	If colour-coding is used, use the colours that stand out.
5	Ensure that multiple floating windows/views of code are distinguishable from one another by visible and noticeable differences in titles.
6	Use a comma to separate items in a horizontal list rather than a space.
7	Make sure that object size is not too small to be noticeable so users do not have to search for it.
8	Allow adequate separation between different parts of a graphical primitive.
9	Make the icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.
10	Use two-dimensional representations as much as possible. If three-dimensional representations must be used, use them effectively.
11	Make windows/views distinguishable from one another by making use of visible and noticeably different icons.
12	Use lower case or sentence case.
Layout	
13	Avoid messy windows. Neat layouts should be achieved easily and quickly.
14	The most current window/view must not cover the one leading to it. They are better side-by-side.
15	Provide a facility to tidy up and straighten links.

Table 6.5 (cont'd) Final checklist and design principles for diagrammatic VPLs

Design principles and their checkpoints	
Familiarity	
16	Use standard symbols and operators (for example, using γ or <> for 'not equal').
17	Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).
18	Use familiar icons and those that match real world objects, e.g. ✓ for ticks, ✗or x for crosses.
19	Use familiar or standard representations for programming constructs and functions.
20	Make manipulation of objects (e.g. resizing) intuitive in both directions for paired operations, e.g., copy/delete, shrink/enlarge.
Meaning/language	
21	Use trigger words, meaningful names or symbols.
22	Use easy language for dialogues, help, text and error messages.
23	Use consistent naming convention (upper/lower case, brackets, quotation marks, etc.).
24	Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity.
Reference	
25	Use numbers or letters as points of reference across windows/views.
26	Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program.
Principle 3: ***Use secondary notation as appropriate***	
27	Use colour-coding and shading as a secondary means to convey a meaning. Use them in a consistent manner.
28	May use colour-coding in labels, names of different categories, types, or groups.
Principle 4: ***Support modification through simplicity, clarity, and flexibility***	
29	Provide a low fidelity tool.
30	Stay simple with fonts – do not use fancy and different font types.
31	Allow users to edit default objects properties such as name and data type.

Table 6.5 (cont'd) Final checklist and design principles for diagrammatic VPLs

Design principles and their checkpoints	
Principle 5: Support evaluation	
32	Provide animation where appropriate such as in debugging tools.
Principle 6: Offload cognitive efforts required where possible	
33	Avoid scrolling or keep it to minimum.
34	Do not use too many colours in the colour-coding scheme for textual messages, titles and names.
35	Avoid complex traversing rules.
Principle 7: Support minimalism and economy of interaction	
36	Remember that too much automation is not good sometimes.
37	Provide an icon for quickly starting a new task such as a new project. That is, make the initial step easy to figure out.
38	Provide icons for some frequently used functions for easy access (undo, execute).
39	Provide an undo function for all operations in manipulating objects (delete, copy, grouping).
40	Automatically adjust the object to an appropriate size.
41	Assign only one primitive to include a few operations/tasks that are frequently used together to do a task.
42	Allow code to be created on the fly – any time; even while the program is running.
Principle 8: Operation on devices should meet user's expectation	
43	Make appropriate use of left and right mouse-clicks for different tasks or functions on the same object (as would be expected by users). Otherwise, it only causes confusion.
Principle 9: Encourage user's control and freedom	
44	Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order for doing things (in creating objects and links, defining attributes, etc.).
45	Users can add comments at any time and anywhere and are free to hide or show the comments made.

Table 6.5 (cont'd) Final checklist and design principles for diagrammatic VPLs

Design principles and their checkpoints	
Principle 10: ***Avoid hard concepts***	
46	Avoid hard concepts that require thinking ahead in: - Passing controls. - Terminating a loop. - Performing iteration. - Referencing. (This is only applicable to Prograph VPL.)
Principle 11: ***Make help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension***	
22	Use easy language for dialogues, help, text and error messages.
47	Use graphics in the HELP document – make it visual.
48	Ensure Help provides a full coverage of all operations and functions.
49	Provide a list of the exact names of operators or fuzzy search facility.
50	Ensure Help does not provide incorrect or outdated information.
51	Provide adequate information in error messages.
52	A previous error message should either disappear or make it known that it is not applicable now.
Principle 12: ***Ensure consistency in provisions (e.g. of functions) and their implementation***	
53	Do not provide a feature or function that is not meant to be available.
54	Make all available features work.
55	Debug the application thoroughly.
Principle 13: ***Ensure consistency in the ways things are done***	
56	Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules – make it applicable in all situations, etc.
Principle 14: ***Prevents or corrects for errors (by providing appropriate automated functionality and by avoiding misleading appearance)***	
57	Avoid misleading users by having a part in the object that looks meaningful but is meaningless or never used.
58	Provide automatic facilities such as error-checking, garbage collector, and spell- check facility.

6.6 Chapter Summary

This chapter provides a synthesis of the theoretical and empirical findings in previous chapters to recommend a checklist and design principles for diagrammatic VPLs. The procedure to generate them is both structured and rigorous. Twenty-seven checkpoints forming the first-pass checklist are generated from the literature. The second-pass checklist consisting of 56 checkpoints are then generated from the first list augmented by the empirical data from the Study units 1 to 6 in this thesis. The second-pass checklist was then refined into design principles in two iterations. The first iteration yields 13 first-pass principles, which are refined further utilising Myers' (n.d.) recommended principles for good programming languages to arrive at 14 second-pass principles and 58 checkpoints in all. Both the second-pass principles and Myers' principles are then evaluated by triangulation with other research. The results suggest a superiority of the principles generated and recommended here over Myers' (n.d.), which were heuristic-based (Nielsen, 1993).

7. SUMMARY AND CONCLUSIONS

7.1 Summary of this research

The main objective of this thesis was to investigate and identify usability problems surrounding Visual Programming Languages (VPLs) in order to produce a checklist and design principles for VPLs that emphasise usability. This chapter provides a summary of the work that has been carried out, its findings and contributions, and discusses potential future research directions arising from the findings of this research.

In Chapter 1, we presented the context of this research and provided some evidence from the literature for the potential in investigating usability issues of VPLs, which went on to assist the formulation of our research objectives. We discussed the multi-disciplinary nature of the research and established its scope, approach, and methodology.

In Chapter 2, we reviewed the literature in Psychology of Programming (PoP), diagrammatic notation, Cognitive Dimensions of Notations (CDs), and *Visual Language* design. Critique and analyses of the theoretical and empirical findings in PoP enabled us to summarise a conceptual model of the psychological process of programming (MoPP, or what is called 'Model of Programming Process' in this thesis). In this model, we identified two major areas and their relevant research questions to be investigated in the research. The two areas examined were the programming paradigm and perceptual coding. Further review and analyses of the literature in diagrammatic notation and CDs helped us establish a set of six design principles for diagrammatic VPLs that would help support their usability. The literature review in *Visual Language* introduced us to applying the concept of the Visual Language Matrix (VLM – a structured framework for the holistic design of text-based documents) to VPL design. As a result of this adaptation, we obtained a VLM for visual programs, which consisted of visual elements that could be used as perceptual cues for visual programs. Both the design principles and the visual elements were later utilised, in Chapter 6 – in conjunction with other findings made during the course of the research – to produce a full set of empirically based principles and a checklist for designing diagrammatic VPLs. In respect to the research questions asked in this chapter, these led to the empirical studies presented in subsequent chapters of this thesis.

Chapter 3 explored one of the research questions raised in Chapter 2, in relation to the programming paradigm and program modality (textual versus visual programs). A within-subjects experiment was conducted in which programmers' performances in a conventional textual program were compared with three equivalent visual programs in control-flow and data-flow programming paradigms. Results revealed the superiority of control-flow over data-flow programs and of visual over textual programs. Our participants performed significant faster in tracing miniature control flow visual programs than with their equivalent data flow visual programs and control-flow textual programs. Furthermore, the data that we obtained from the pre-test questionnaires indicated a control-flow preference among our participants – first year undergraduate students. This adds evidence for a control-flow bias indicated in the literature (Good, 1999). There was also evidence, on the basis of our experiments that those who learned control-flow programs before data-flow programs performed better than those who learned data-flow programs first. Implications, from the results, for designing VPLs for usability were also discussed.

Chapter 4 presented four experiments that investigated the effect of directional representation and of traversal direction on novices' comprehension in visual programs. In the former, we compared three directional representations that were commonly used to indicate the flow of programs: arrow, line, and juxtaposition. The two experiments conducted revealed that an arrow was the best and juxtaposition was the most error-prone representation. To investigate the effect of traversal direction, we conducted two experiments that investigated the effect of both traversal direction and directional representation (arrow and line) concurrently. The first experiment, a within-subjects design, compared three different traversal directions: Top-Down, Hierarchical-Nested, and Free-Style. In the second experiment, a mixed-factorial design, we compared five traversal directions: Top-Down, Hierarchical-Nested, Bowles, and two other Free-Styles called Curvy-Net and Rectangular-Net. The second experiment was conducted to avoid the differential carry over effect that was observed in the first experiment. The results from the second experiments revealed the effect of traversal direction on the programmers' performances but not of directional representation. It was found that participants performed significantly poorer when traversal direction had the '*fall back*' feature (a restrictive way in which programs must be traced, described on page 47) than when it did not. These are Hierarchical-Nested and Bowles representations. This provides evidence for Green's (1982) speculation that '*fall back*' imposed cognitive demands on programmers. In short, the second experiment revealed a clear-cut conclusion on the issue of representation of program flow. Firstly, arrow and line made no difference on the programmers' performances and secondly, traversal direction that had the '*fall back*' feature adversely affected programmers' performance because it made the program harder to trace.

Aside from what we intended to investigate (i.e representation of program flow), further analysis of the data obtained provided evidence supporting the Match-Mismatch hypothesis for VPLs as had been reported for the second experiment in Good's (1999) thesis. Due to the contradictory findings in the literature in this respect (those by Curtis *et al*, 1989; Moher, 1993; and Good's (1999) first experiment), our finding and Good's (1999) second experiment provide evidence that some knowledge derived from research findings in PoP was not limited to textual programs, but applicable to visual programs as well.

Chapter 5 presented a holistic evaluation of the commercial VPL, Prograph. The purpose of this study was to obtain a list of usability problems found in learning and using Prograph. Therefore, this study offered a much more comprehensive but less detailed coverage for potential usability problems to be found in VPLs than the experimental approach taken in the previous two chapters.

We first looked at a variety of evaluation methods traditionally used by HCI practitioners and researchers in programming language design to find a method that would be most appropriate for evaluating a programming language. Following our investigations, the Cognitive Dimensions of Notations (CDs) method was found to be the most appropriate, despite some weaknesses. Our critique and analysis suggested restructuring the evaluator's analysis space while carrying out an evaluation as a means to improving the usability and reliability of the method. This in turn raised another research question for the Prograph study. In addition to a list of usability problems, we also wanted to find what usability problem areas existed in the domain.

Different research methods were then investigated as to how Prograph could best be evaluated in order to provide the answers to the two research questions that we had raised. Nothing was found to be practical in the HCI toolbox, so we turned towards methods that were not conventionally used by HCI practitioners. The study used a combination of two methods from the social sciences: immersion and auto-observation. Immersion is a method commonly used by sociologists and product designers whereby the researcher lives experience of the product users or of the people who are the subject of the researcher's interest. Auto-observation is a method used by existential sociologists whereby the researcher observes himself/herself whilst in participant role. In this study, we used the diary technique was for data collection.

Content analysis of the diary revealed 145 usability problem tokens covering ten problem areas. These problems were later used in the Chapter 6 in conjunction with other previous findings in the research. However, the ten problem areas comprising the Prograph usability problems were analysed further. Pareto analyses were conducted for the problem areas found and for the dimensions in CDs violated, in order to prioritise the problem areas and dimensions to be considered. This yielded a proposed extension to the procedure to carry out CDs analysis for the evaluation of Prograph in later versions if needed. The chapter concluded with two empirical studies that demonstrated the applicability of the outcomes of the Prograph study to a different research context. The studies extended heuristic evaluation (also a predictive and inspection evaluation method like CDs) by incorporating a set of usability problem areas commonly found for a product type (called '*Usability Problems Profile*', a concept introduced as a result of the Prograph study) to its procedure. This extended method was called HE-Plus. The studies showed the superiority of HE-Plus over heuristic evaluation in terms of reliability of the results and of usability.

In Chapter 6 we brought together the findings from previous chapters, both through review and analyses of the literature and through the empirical studies we had conducted during our research. We described a structured process of deriving a checklist and principles for VPL design from the research undertaken in the preceding chapters. There were three phases to this process: formation; refinement; and evaluation. In the first phase (formation), the first-pass checklist consisted of 27 checkpoints were formed from the six principles for designing diagrammatic notation, which we derived from the literature in Chapter 2, and the VLM for visual programs, which we adapted from Kostelnick & Roberts' (1998) VLM for textual documents – also described in Chapter 2. The first-pass checklist was then checked against the findings from our experiments (Chapters 3 and 4) and the 145 usability problem tokens identified in the Prograph study (Chapter 5). This resulted in the second-pass checklist consisted of 56 checkpoints. In the second phase (refinement), 13 first-pass principles were obtained from this second list and compared against the only published set of principles for programming languages available (Myers, n.d.), but which were not obtained empirically. This phase resulted in 14 second-pass principles in all. The last phase was evaluation. We evaluated the second-pass principles and Myers' (n.d.) set of principles against the findings of two usability evaluations discussed in the literature. One evaluated GUI-based programming languages and environments. The other evaluated Prograph using CDs analysis. Ideally, if the two sets of design principles – ours or Myer's (n.d.) – should be comprehensive enough, we should be able to find at least one match between the problem and the design principles. In other words, at least one of the principles in the sets would account for each usability problem reported by the two usability evaluation studies. The evaluation resulted from this matching process indicated that our design principles could account for all the problems found in the two usability evaluation studies; while Myers' (n.d.) principles could not.

7.2 Conclusions

This research has made original contributions to knowledge in the fields of HCI, PoP, and VPLs as follows.

7.2.1 Major findings: Design principles for VPL designers

The main objective of this research has been progressed through the derivation of a checklist and principles that are comprehensive and based on empirical data obtained either in this research or from the literature. Designers of VPLs, particularly those of diagrammatic type, now have access to an empirically grounded set of design principles that put an emphasis on usability. The usefulness of the checklist, however, is expected to be more specific to a VPL that has similar characteristics to Prograph. It is up to the designers to consider tradeoffs between checkpoints and the principles given in the thesis, as appropriate to the application or language being developed. Furthermore, we also hope that these principles and checklist can, to some extent, help them devise their own in-house heuristics or style guide.

7.2.2 Empirical evidence contributing to novel knowledge

The following findings resulted from the experimental studies conducted in this research. Items 1 to 3 directly answered our research questions while others were by-products resulting from the analysis of various forms of data collected during the course of the experiments. These findings are:

1. Traversal direction affects the programmers' performances and too much structure and too many rules imposed on readers of diagrams might only increase cognitive load and decrease diagram usability. This evidence might be used to support an argument against any attempt for a rigid design of diagrammatic notation in the future.
2. Using an arrow or a line as a representation for program flow does not affect the programmers' performance.

3. Tracing control-flow visual programs is easier than tracing data-flow visual programs. Novices' performance was significantly faster with control-flow visual programs than data-flow visual programs.
4. Research findings, which are based upon textual programming languages might also be applicable to visual programming languages. This is because our research as well as that of Good's (1999) provide evidence – for visual programs – the Match-Mismatch phenomenon that is commonly observed for textual programs.
5. Learning a control-flow language first might facilitate transfer from learning control-flow concepts to learning data-flow concepts better than transfer from learning data-flow concepts to control-flow concepts.
6. There is a control-flow bias among first year students. Our questionnaire data revealed the highest percentage of procedural programming languages being known by our participants by the time they started their first year at a university. This observation was consistent across three universities participating in our studies.

7.2.3 New Methodology: A new framework to CDs for evaluating diagrammatic VPLs

The Prograph study yielded an extended framework to CDs as a method to be used for evaluating later versions of Prograph. It is proposed here that the approach to this extension be applied to different products (e.g. applications, languages – using the new framework to CDs analysis) or to other inspection methods as well (e.g. restructuring the procedure to carry out method concerned). Direct evaluation of the extended framework was not possible within the time frame of this research. However, the approach of this new framework has been supported with empirical evidence provided by two different studies has been described in Chapter 5.

7.3 Limitations

7.3.1 The Prograph study

The pragmatism exhibited in the research (Study unit 6 – Prograph study) opens the thesis up to criticism because everything was carried out by one and the same person – from design, data collection, data interpretation, to data analyses. As such, the limitations of the research are discussed below.

Generalisability

At a glance, one could argue that the findings of this study cannot be generalised because the study did not employ a quantitative research method that used inferential statistics. The Prograph study adopted the naturalistic inquiry approach, in which genralisability refers to transferability of findings between similar sending and receiving contexts (Lincoln & Guba, 1985). It is therefore transferability in the context of the naturalist's interpretation that is relevant and important. Transferability of this research has, in fact, been demonstrated through other research carried out by Chattratichart & Brodie (2002a & 2002b). This was discussed in Chapter 5 and in Section 7.2.3.

Credibility of data interpretation: how was bias dealt with?

In this study the researcher was the *novice programmer*, the *evaluator* (of Prograph), the *documenter*, and also, the *data interpreter*. How, then, could bias have been avoided in data interpretation? Admittedly, bias could not possibly be avoided. We, however, argue that it is better to compromise by considering the tradeoffs between using an outsider and using the documenter to do data interpretation, i.e. adopting the more favourable or less harmful alternative.

Firstly, user experience is a subjective matter. It involves the users' emotions arisen from pleasures with using the products or from successes in accomplishing a certain task, in mastering a difficult concept or in finding a persistent programming bug. On the other hand, failure to do a particular task, to understand difficult a programming concept or unfamiliar construct, or to find help information at the time it is needed, causes frustration. Usability problems of Prograph caused these failures, which in turn leads to frustration and poor user experience. Experience or emotions are not easy to quantify and, worse, to empathise with. It would therefore be difficult to establish a benchmark to measure the correctness of the interpretation of the data that involve users' experience (e.g. joy, frustration, boredom).

Secondly, knowing contexts of use plays an important role in analysing the content of the diary. In light of the multi-tasks that the documenter was doing at the time of documenting, it was highly unlikely that she would record all minute details of the interface and of her experience in precisely and detailed enough so that there is only one way to interpret the content. Lack of contexts adds to the difficulties of obtaining accurate interpretation by an outsider.

To summarise from the above, interpretation by an outsider might well be less accurate than that by an insider who immersed herself in the learning and using Prograph and therefore knew the context of use well and is likely to empathise with the user (herself) better than an outside interpreter. In short, using an insider to interpret the data could yield us biased interpretation whilst using an outsider could yield incorrect interpretation. Further, there would be no guarantee that interpretation by an outsider was free from bias from his/her prior domain knowledge or system of beliefs.

Thus, rather than debating about biased interpretations of the data, which could have resulted in a deeply unproductive inquiry, it would be more fruitful to demonstrate the credibility of our findings through triangulation of our findings with those in the literature and through a demonstration of their transferability to other research context. Triangulation of Prograph findings with the findings by Houde & Sellman (1994) and by Green & Petre (1996) has been discussed in Chapter 5. Transferability of the outcomes of this study to web site evaluations by extending the heuristic evaluation method has also been demonstrated and discussed in Chapter 5.

Evaluation of the extended framework to CDs

As a result of the Prograph study in Chapter 5, we proposed an extended framework to CDs to be used as an evaluation method for diagrammatic VPLs. It would have been ideal to test the framework by having a few evaluators carrying out a CDs analysis on Prograph, using the extended framework and see whether it would be easier to do CDs analyses than using the original procedure (i.e. without the CDs profile or '*Usability Problems Profile*') and whether evaluators would find problems outside the profiles used. This, unfortunately, was not easily operational. Firstly, there were no Prograph learners who would also able to do CDs analysis. Furthermore Prograph is not a learning language. Learners of Prograph were more likely to be professional programmers who needed to or wanted to learn Prograph for their jobs. It would be unlikely that these professional programmers would also be an expert in CDs. They would need to be trained to do CDs analysis. As reflected by her experience report, Kutar (2000) stated that CDs technique was not easy to learn and practice. Training programmers would therefore require more than just a few hours of their time or even a day. Due to difficulties in recruiting and resource constraints, especially at the end of this research project when resources had already been exhausted both in terms of time and budget, evaluation of the proposed extended framework was hence impossible to do and left as a subject of future research.

7.3.2 The experiments

Despite its disadvantages, as discussed in Chapter 5, the experimental method was chosen as the most suitable research method for some of the research questions we wanted to investigate. The programs used in the experiments oversimplified real programming situations in order to control for confounds. What the participants in the experiments saw were only static snapshots of a very simple and small section of a program. It is therefore questionable whether we would observe similar effects in real-life programming contexts and how serious these would be in relation to other co-existing factors that also adversely affect programmers' performance. Another concern is the extent to which the findings from these experiments with students could be extrapolated to expert programmers. Nonetheless, we had established earlier in Chapter 1 that we would focus on novice programmers in this research. Therefore, the issue could easily be addressed in future research by interested parties.

7.3.3 Checklist and principles

The checklist and principles we derived in Chapter 6, however carefully refined and evaluated, have not been tested or used in real-life situations. This is inevitable considering the time frame of the research and the resources available and is, therefore, a potential subject of future research as suggested in section 7.4.2.

7.4 Future research

7.4.1 VLM for visual programs

In Chapter 2 we derived the VLM for visual programs based upon an analysis of the VLM for text-based documents. The VLM could be improved further by re-evaluating the visual elements in the VLM against findings of relevant VPL or usability research, and/or by conducting experiments to test hypotheses about certain visual elements in the VLM. A comparative study could also be carried out to test the improved version of VLM at a later stage.

7.4.2 Checklist and principles

As mentioned in Section 7.3.3, the checklist and principles we recommended in this research still need to be tested. A possibility for future research is to carry out a comparative study between a group of VPL designers using the checklist and principles derived here and another group not using the principles but possibly adopting a different set of design principles and heuristics (as determined by the researcher of the study). Although comparisons will be made (between the two groups), it is envisaged that a tightly controlled experiment would not be possible or appropriate. A mixed methodology employing both qualitative and quantitative methods might have to be followed. For a fair comparison, certain levels of control might have to be imposed such as project deadline, number of designers in each team and their experience, progress monitoring scheme, amount of time the team members spend to do the design, etc. Data collection might include video recording of the design activities throughout the project life, artefacts produced at different stages, evaluation results (if carried out during the project lifecycle). Data of the evaluations during the design lifecycle might be obtained by ways of user testing, interviews, questionnaires, etc. If co-operative evaluation (where the user and the facilitator go through the application together) was to be chosen as more appropriate than user testing, video or tape recording would deem necessary.

7.4.3 The extended framework to CDs

The extended framework to CDs that we proposed could not be tested within the time frame of this research. It is therefore recommended that in the future, this new framework should be tested to see whether it would be easier for evaluators and whether it would yield reliable results, i.e. a small number of false alarms and reasonable overlapping results obtained by different evaluators, and whether it can be used for other VPLs as well.

7.4.4 Profile bank

The kernel of the extended framework to CDs proposed in Chapter 5 is that adding the '*Usability Problems Profile*' as another layer to the existing procedure of usability evaluation methods that are predictive in nature, will improve the reliability of the evaluation results and, possibly, ease of use of the method. The two HE-Plus studies described in Chapter 5 in which we compared heuristic evaluation to HE-Plus (an extended method to heuristic evaluation using a '*Usability Problems Profile*') indicated that a profile existed for web applications and helped ease the original method. This merits future research. Our question here is whether profiles do really exist and what they are for different types of applications. In the presentation at HFES 2002 conference, the author of this thesis called for further research and collaboration between academics and the industry to compile a '*profile bank*', which is a database of problem areas for different types of applications, so that evaluators can, in the future, choose an appropriate profile for what needs to be evaluated.

7.4.5 A method for usability evaluation of complex systems

Immersion and diary techniques were used in our Prograph study. Despite the limitations of the study (discussed in Section 7.3.1), its findings transferred well to a different research setting as discussed and demonstrated by the two HE-Plus studies in Chapter 5. One venue for future research is, therefore, further investigating the use of these two techniques in developing a usability evaluation method for complex systems (such as programming languages), which cannot be easily and holistically achieved using conventional HCI methods.

APPENDIX A

Paradigm Study
(Chapter 3)

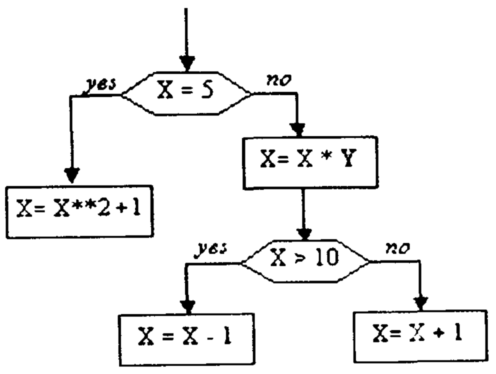
A-1: Textual Program Used for the Experiment in Chapter 3

```
If S =  '*Bad*' then
    If S = '*Pretty*' then
        Loop begins for Times = 1 to 2
            If S = '*Sad*' then
                Print 'Shout'
            Else
                Print  'Goal'
            End if
        End loop
    Else
        If S = '*Funny*' then
            Print 'Nod'
        Else
            If S = '*Sad*' then
                Print 'Goal'
            End If
        End If
    End If
Else
    If S = '*Sleepy*' then
        If S = '*Pretty*' then
            Print 'Wink'
        Else
            Print 'Shout'
        End if
    Else
        Loop begins for Times = 1 to 2
            If S = '*Funny*' then
                Print 'Nod'
            Else
                Print 'Wink'
            End if
        End loop
    End if
End If
```

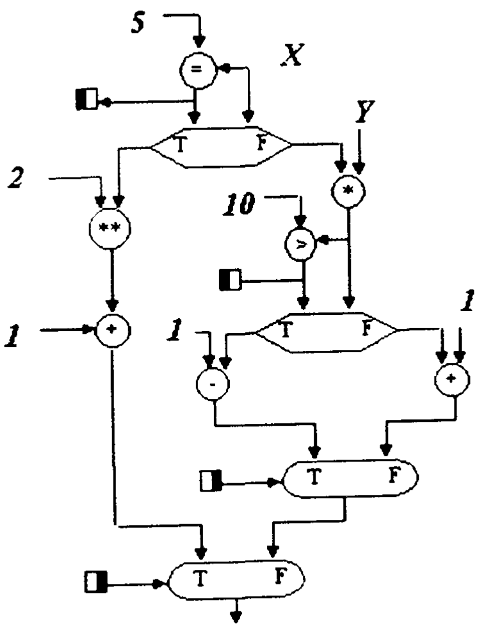
A-2: Visual Program Samples

Six program samples in three traversal directions and two paradigms are given here as follows:

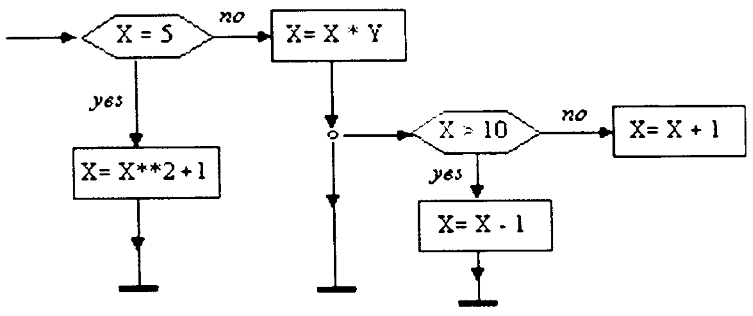
1. Top-down, control-flow



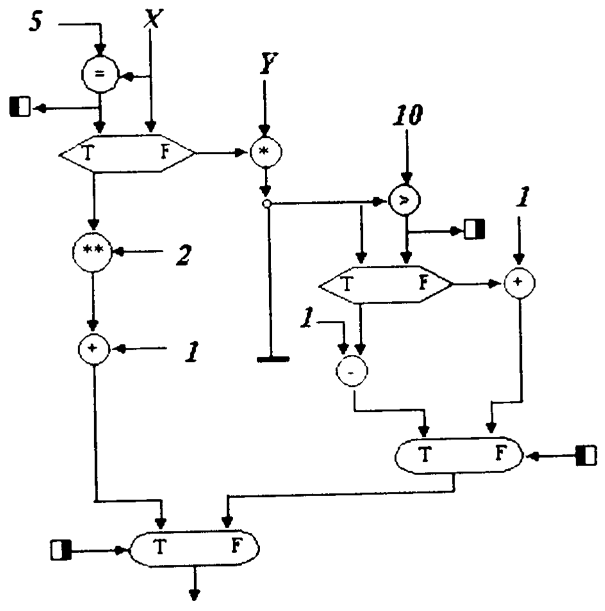
2. Top-down, data-flow



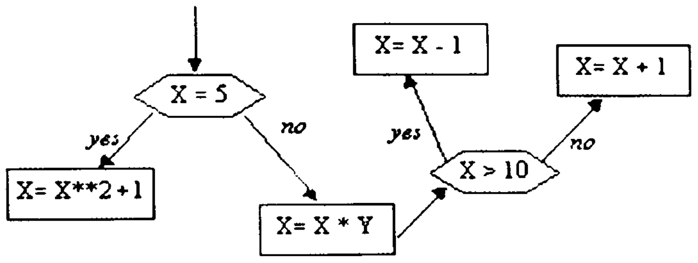
3. **Hierarchical-nested, control-flow**



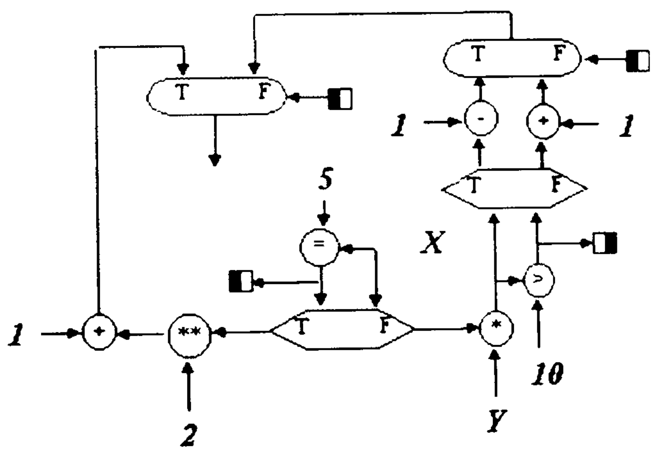
4. **Hierarchical-nested, data-flow**



5. Free-style, control-flow

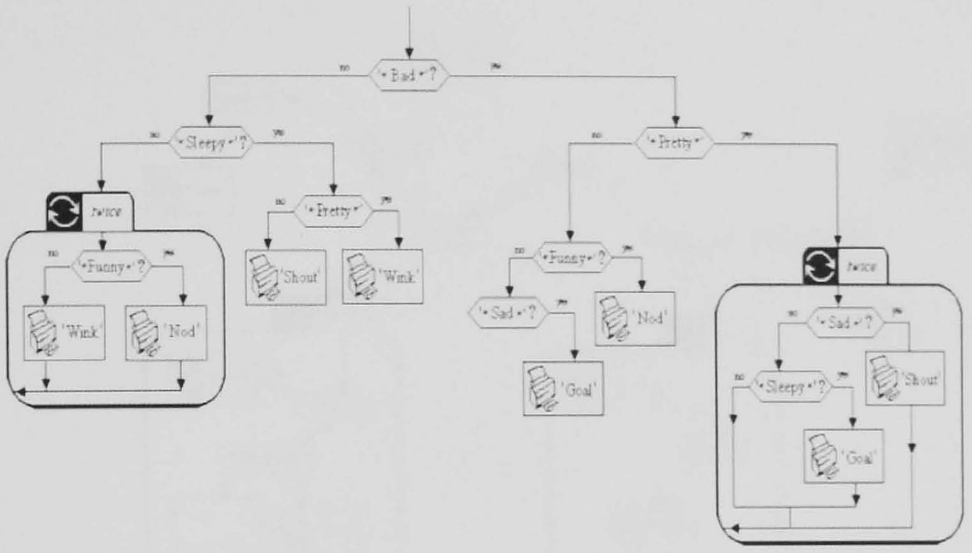


6. Free-style, data-flow

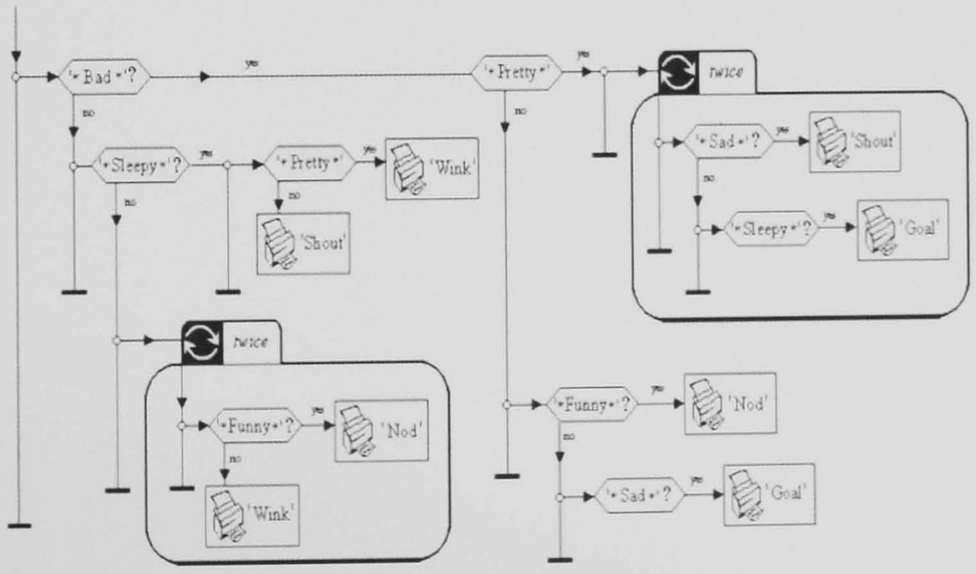


A-3: Control-Flow Visual Programs Used

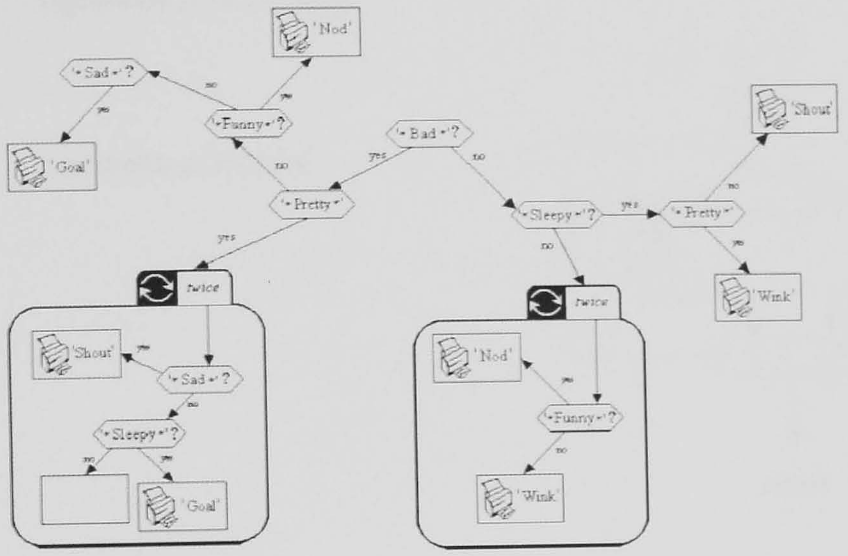
1. Top-down



2. Hierarchical-Nested

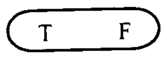


3. Free-style


'Nod'
'*Sad*'?
yes
no
'Goal'
'*Funny*'?
'*Bad*'?
'*Pretty*'
'Shout'
'*Sleepy*'?
'*Pretty*'
'Wink'
twice
'Shout'
'*Sad*'?
'*Sleepy*'?
'Goal'
twice
'Nod'
'*Funny*'?
'Wink'

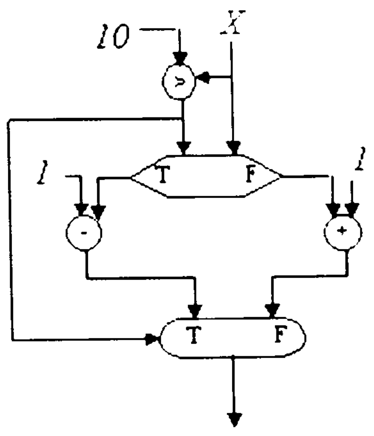
A-4: Distributor and Selector Nodes in a Conditional Construct.

represents a Distributor.



represents a Selector.

Conditional construct



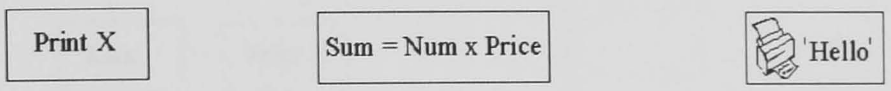
result

A-5: Control-Flow Programs Used in Training Participants

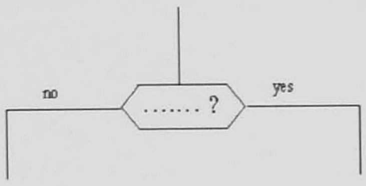
The following are some sample control-flow programs (both textual and its equivalent visual programs) used during the training session before the experiment in Chapter 3 was carried out. Participants were introduced to different graphical representation for basic programming constructs such as If-Then and Loop.

1. Representational Constructs used in the visual programs

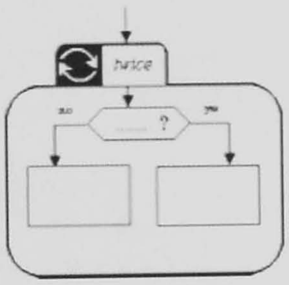
- A statement or expression is represented by a rectangle. Examples are:

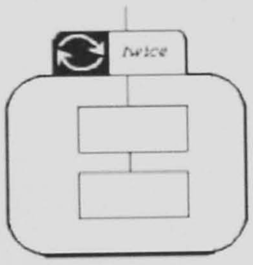


- If-Then Construct



- Loop Constructs



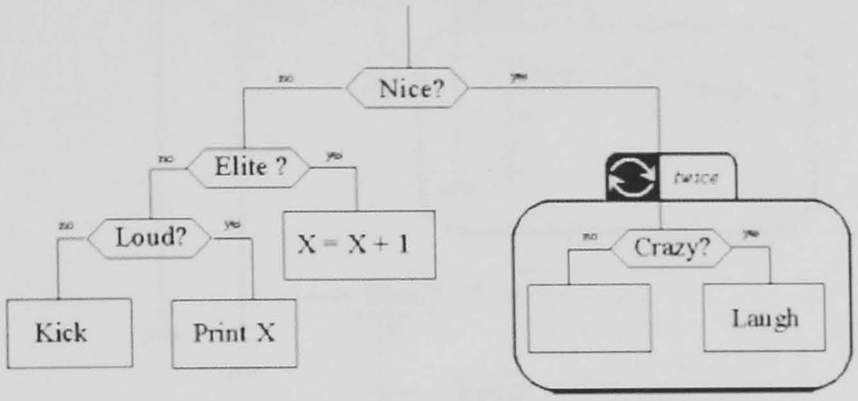


2. Sample of a textual program

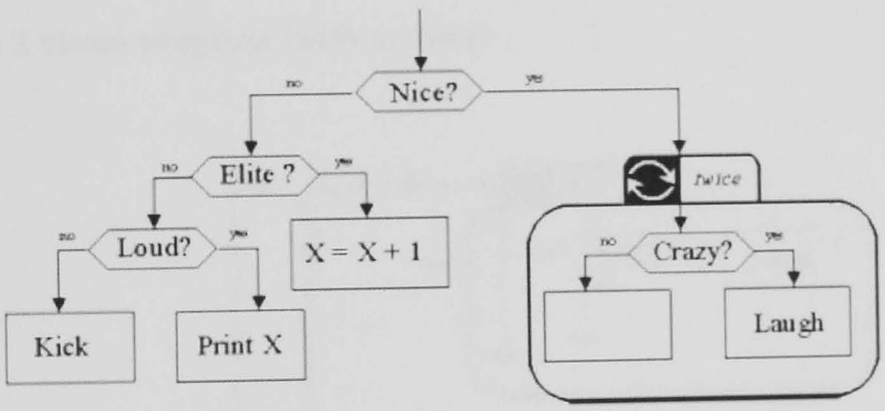
Let S be a string such as 'He is Crazy and Loud!' and the outcome is that X gets printed.

```
If  S = ' *Nice* '  then
  Loop begins for Times = 1 to 2
        If  S = ' *Crazy* ' then
              Laugh
        End if
  End loop
Else
  If  S = ' *Elite* '  then
        X = X + 1
  Else
        If  S = ' *Loud* '  then
              Print  X
        Else
              Kick
        End if
  End if
End if
```

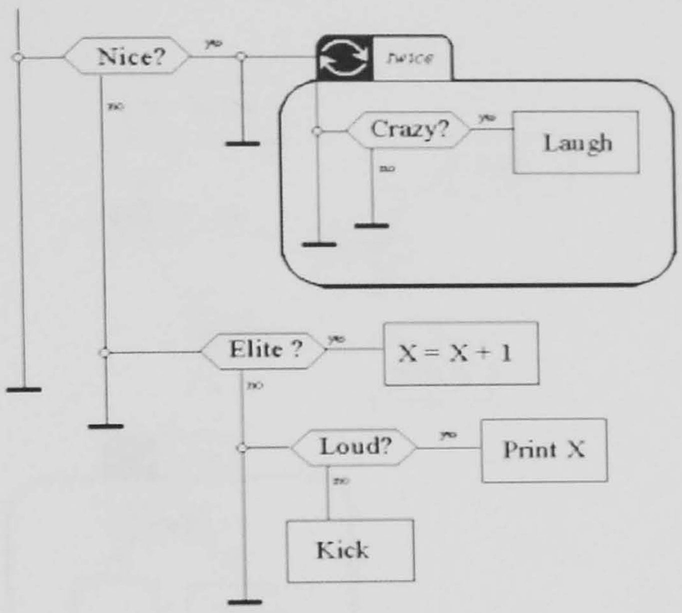
3. **Type 1 visual program (without arrows)**



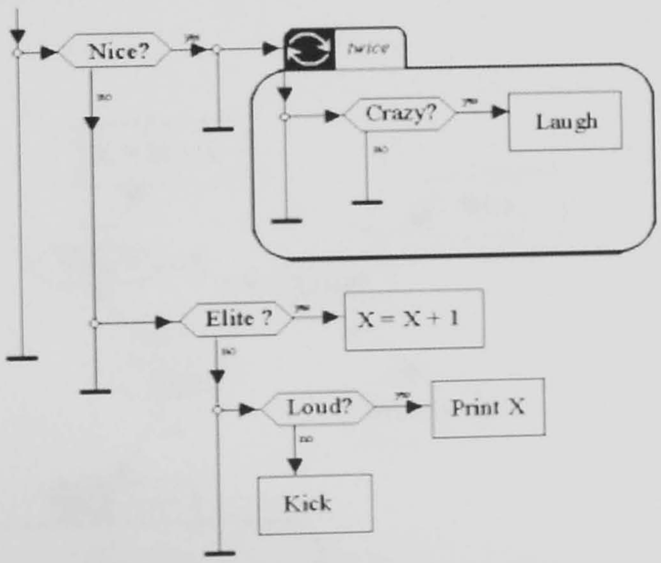
4. **Type 1 visual program (with arrows)**



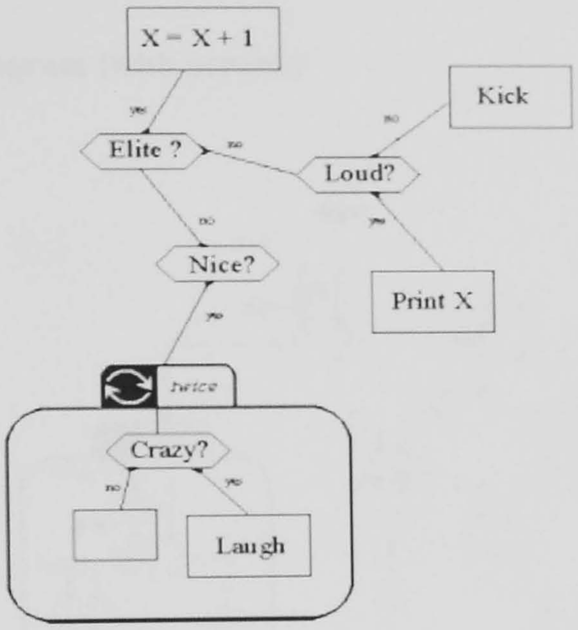
5. **Type 2 visual program (without arrows)**



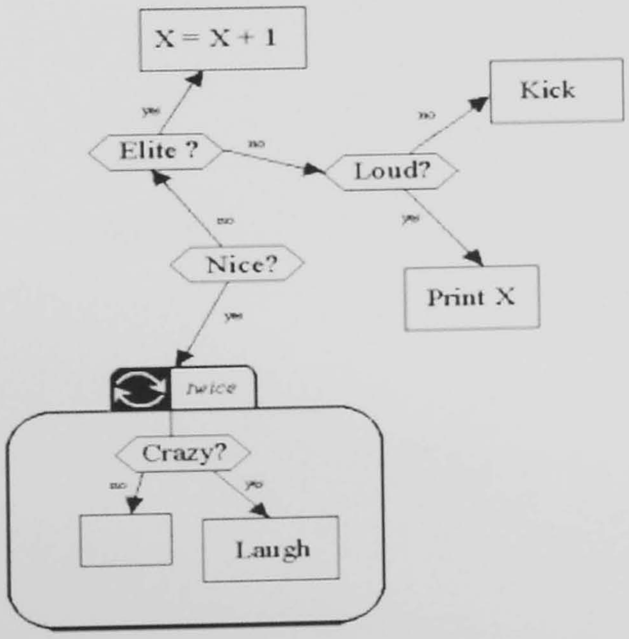
6. **Type 2 visual program (with arrows)**



7. **Type 3 visual program (without arrows)**



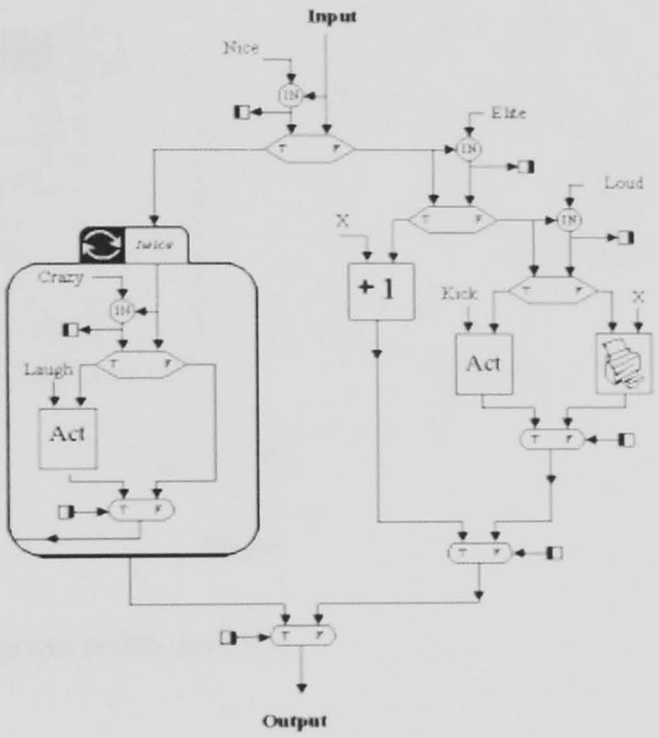
8. **Type 3 visual program (with arrows)**



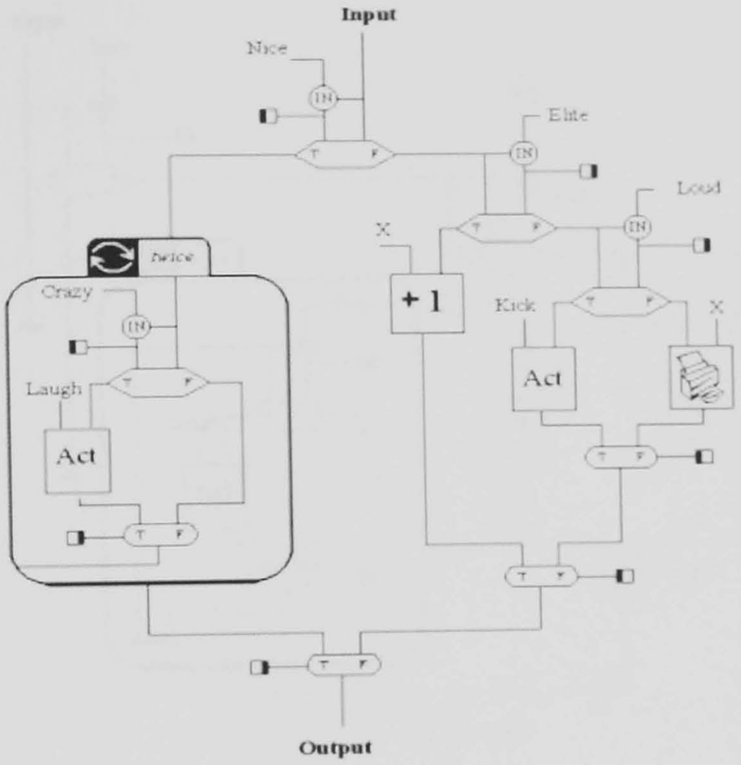
A-6: Data-Flow Programs Used In Training Participants

The following are some sample data-flow programs for the textual program used in Appendix A-5.

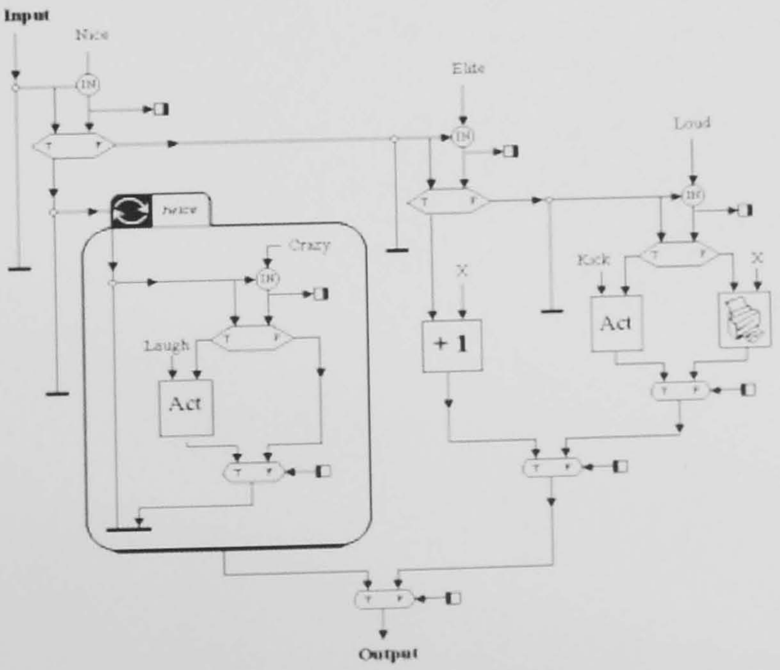
1. Type 1 visual program (with arrows)



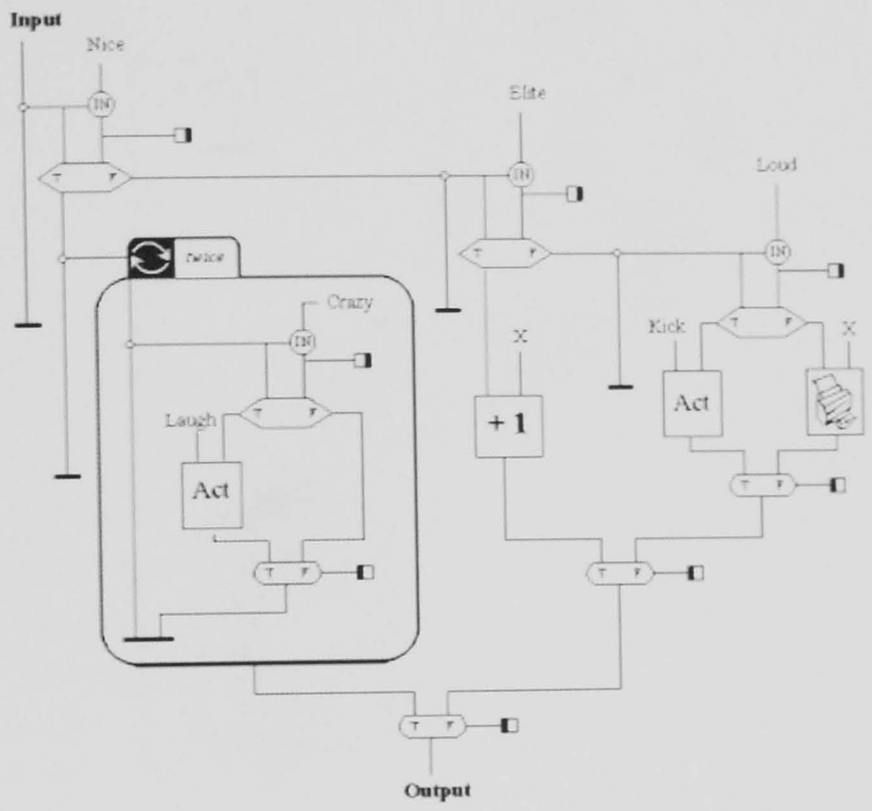
2. **Type 1 visual program (without arrows)**



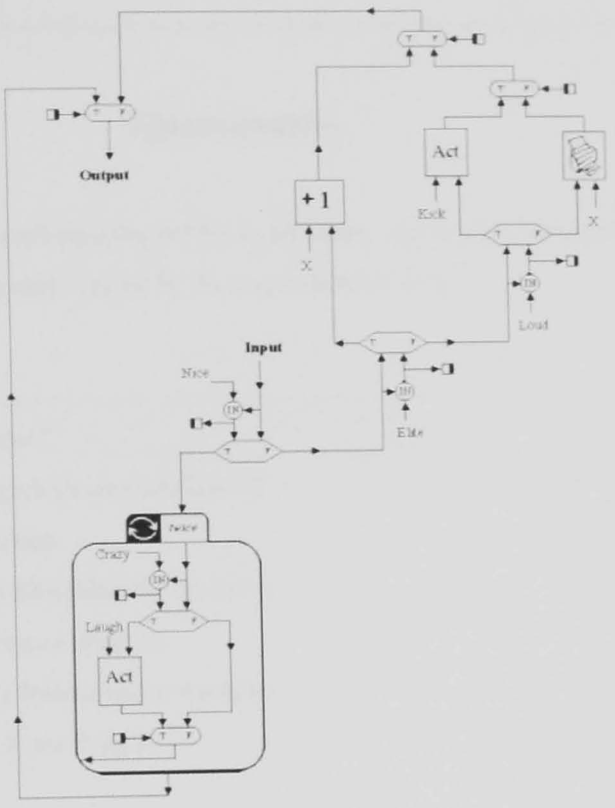
3. **Type 2 visual program (with arrows)**



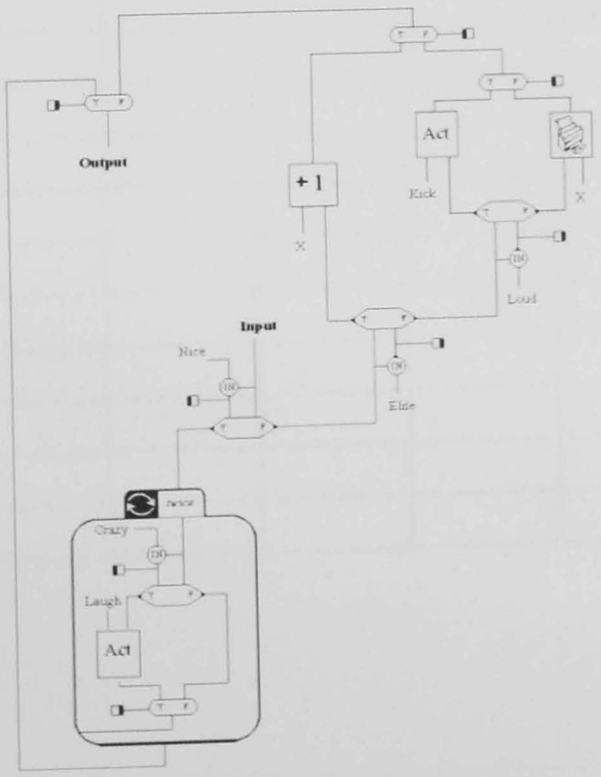
4. Type 2 visual program (without arrows)



5. **Type 3 visual program (with arrows)**

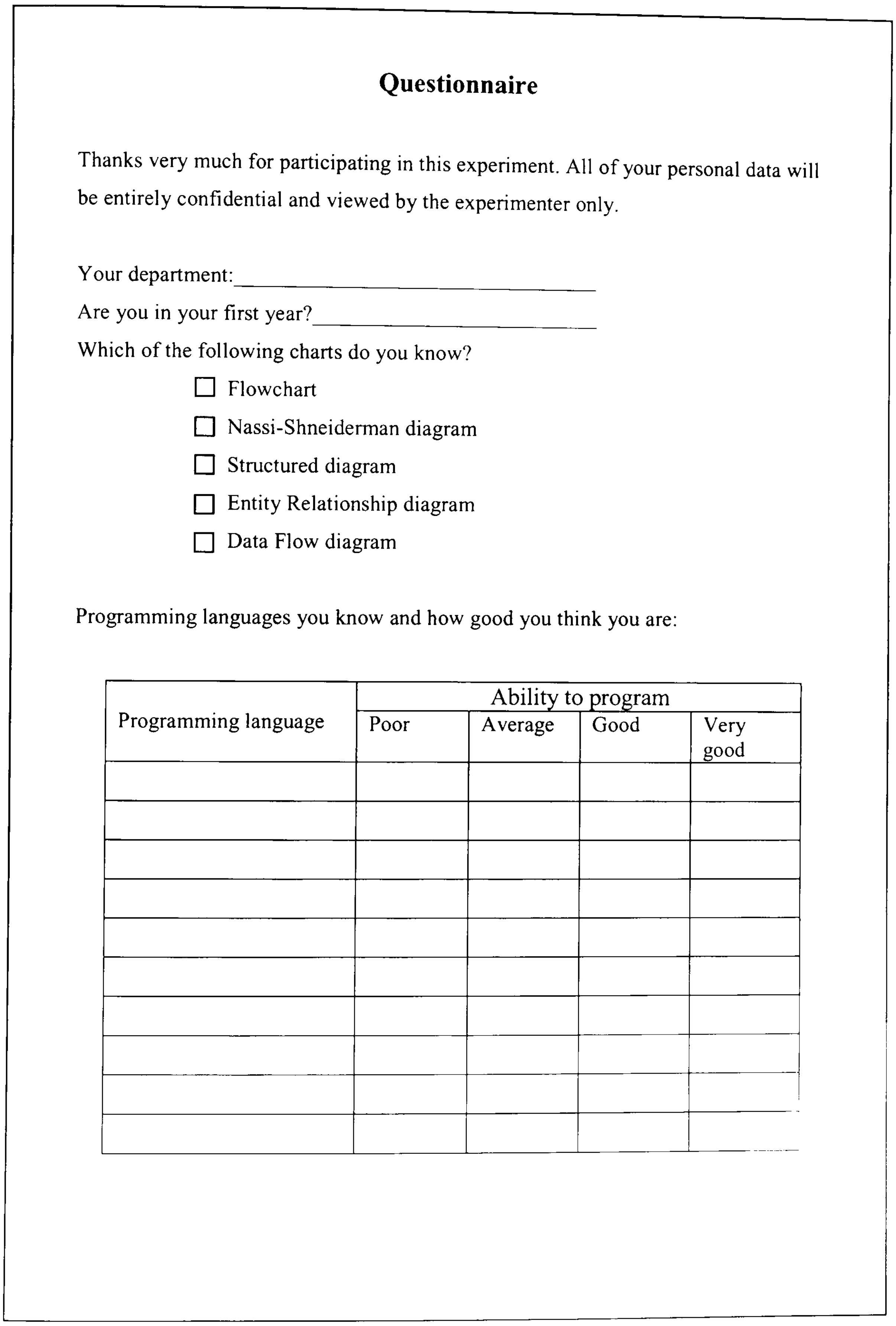


6. **Type 3 visual program (without arrows)**



A-7: Pre-Test Questionnaire

This questionnaire was given to participants of the experiment in Chapter 3.



Questionnaire

Thanks very much for participating in this experiment. All of your personal data will be entirely confidential and viewed by the experimenter only.

Your department:______________________________

Are you in your first year?_________________________

Which of the following charts do you know?

- ☐ Flowchart
- ☐ Nassi-Shneiderman diagram
- ☐ Structured diagram
- ☐ Entity Relationship diagram
- ☐ Data Flow diagram

Programming languages you know and how good you think you are:

Programming language	Ability to program			
	Poor	Average	Good	Very good

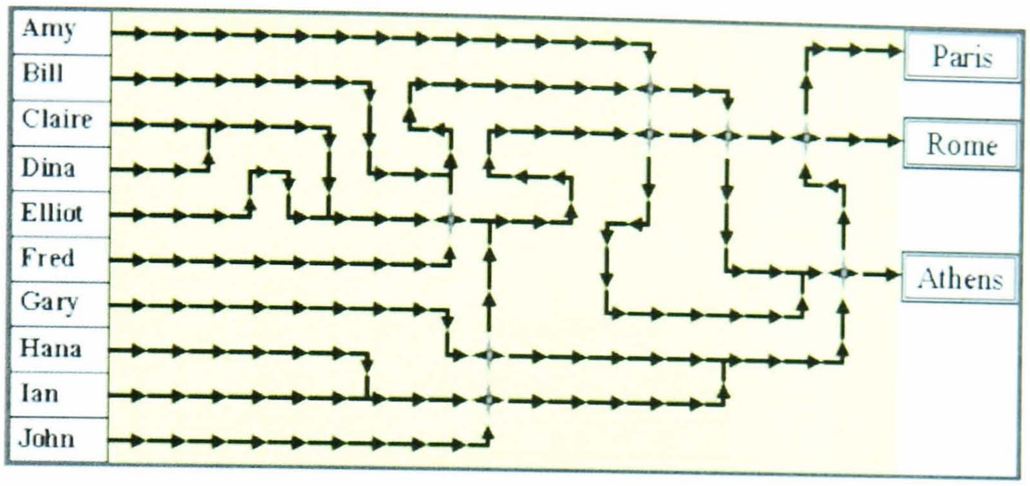
APPENDIX B

Representation of program flow

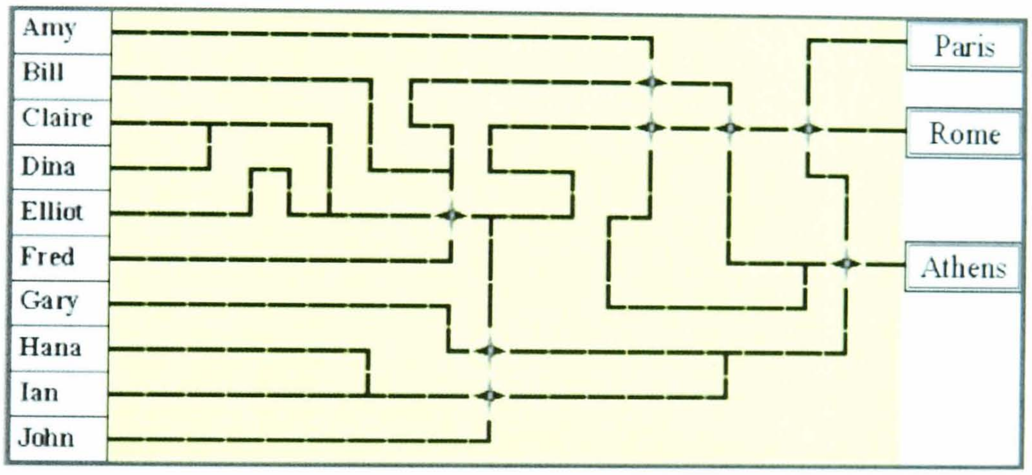
(Chapter 4)

B-1: Maze Studies

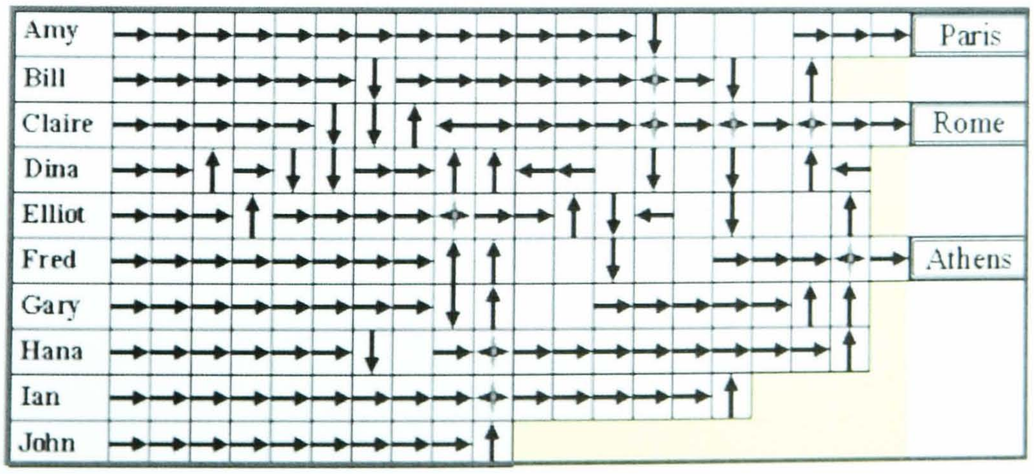
1. Arrow maze



2. Line maze



3. Juxtaposition maze

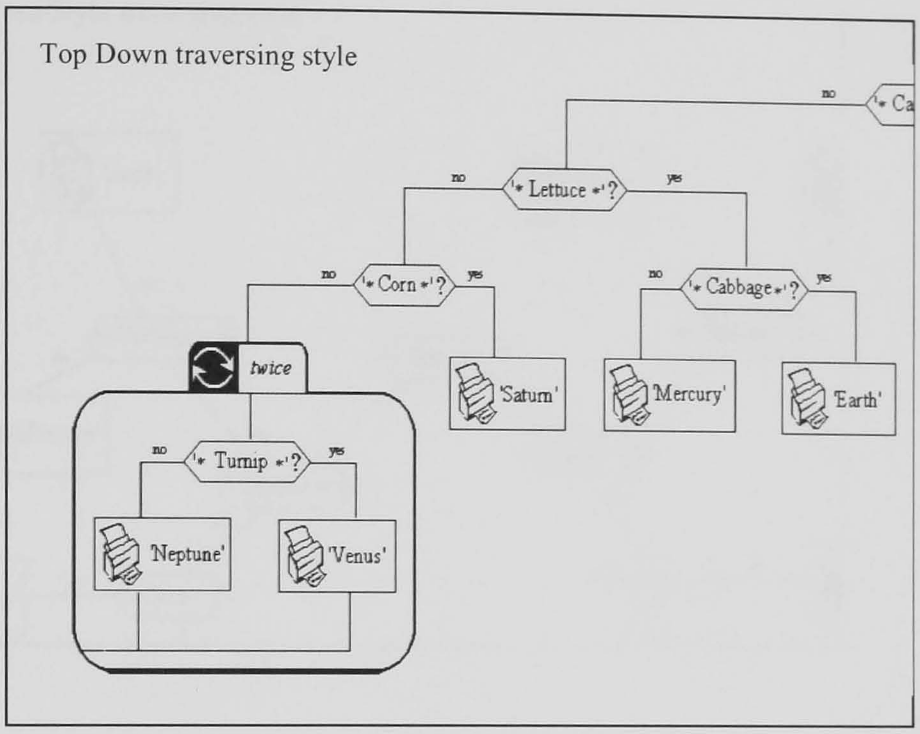


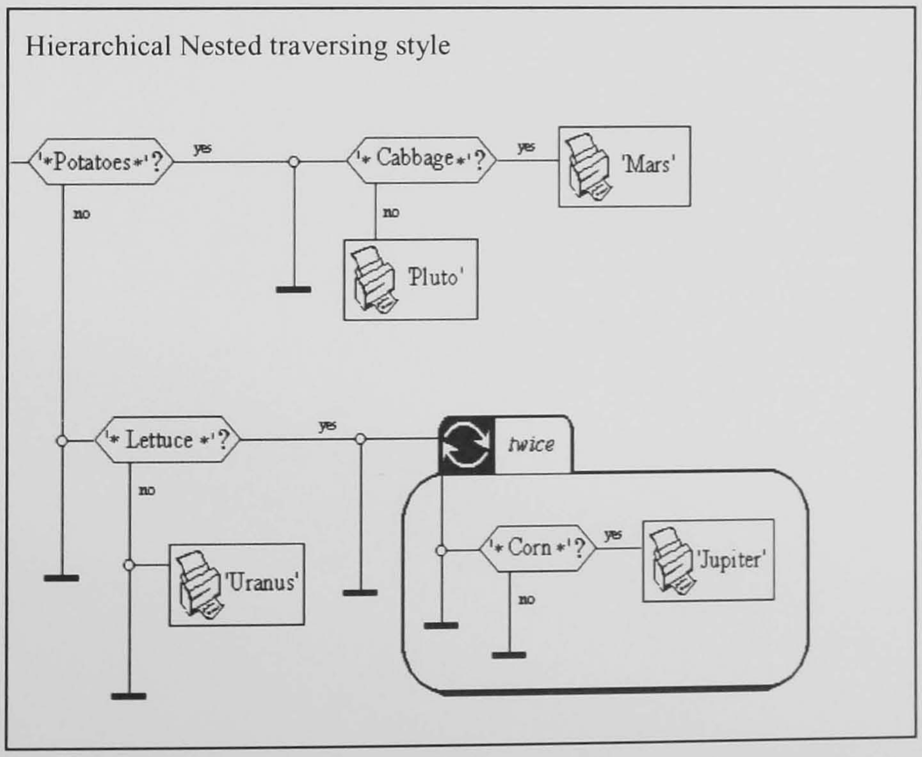
B-2: Flow Study 1

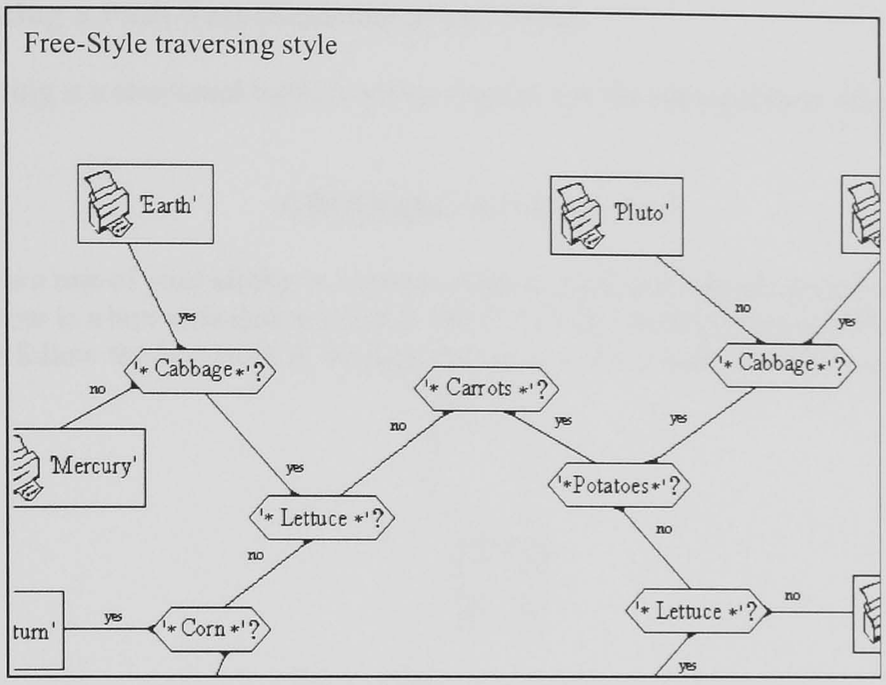
1. Textual program used in the real test of Flow Study 1

```
If S = '*Carrots*' then
    If S = '*Potatoes*' then
        If S = '*Cabbage*' then
            Print 'Mars'
        Else
            Print 'Pluto'
        End If
    Else
        If S = '*Lettuce*' then
            Loop begins for Times = 1 to 2
                If S = '*Corn*' then
                    Print 'Jupiter'
                End If
            End Loop
        Else
            Print  'Uranus'
        End If
    End If
Else
    If S = '*Lettuce*' then
        If S = '*Cabbage*' then
            Print 'Earth'
        Else
            Print 'Mercury'
        End If
    Else
        If S = '*Corn*' then
            Print 'Saturn'
        Else
            Loop begins for Times = 1 to 2
                If S = '*Turnip*' then
                    Print 'Venus'
                Else
                    Print 'Neptune'
                End If
            End Loop
        End If
    End If
End If
```

2. Sample partial programs used for Flow Study 1






Free-Style traversing style
'Earth'
'Pluto'
'Mercury'
turn'
'* Cabbage *'?
'* Carrots *'?
'* Cabbage *'?
'*Potatoes*'?
'* Lettuce *'?
'* Corn *'?
'* Lettuce *'?
yes
no

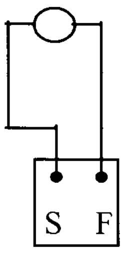
B-3: Flow Study 2

1. Choosing a Path Test (Ekstrom *et al.*, 1976)

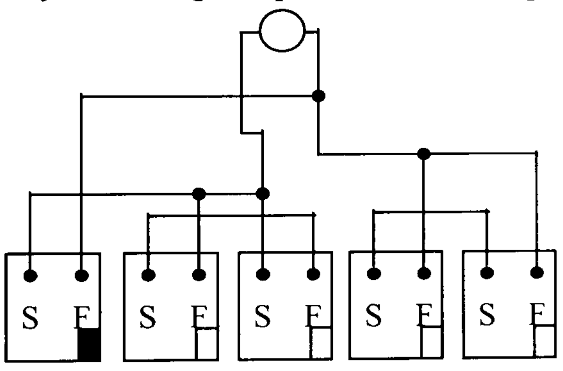
The following is a shortened version of the original test for demonstration purpose.

CHOOSING A PATH -- S-2

This is a test of your ability to choose a correct path from among several choices. In the picture below is a box with dots marked S and F. S is the starting point and F is the finish. You are to follow the line from S, through the circle at the top of the picture and back to F.



In the problems in this test there will be five such boxes. Only one box will have a line from the S, through the circle, and back to the F in the same box. Dots on the lines show the only places where connections can be made between lines. If lines meet or cross where there is no dot, there is no connection between the lines. Now try this example. Show which box has the line through the circle by blackening the space at the lower right of that box.



The first box is the one which has the line from S, through the circle, and back to F. The space lettered A has therefore been blackened.

Each diagram in the test has only one box which has a line through the circle and back to the F. Some lines are wrong because they lead to a dead end. Some lines are wrong because they come back to the box without going through the circle. Some lines are wrong because they lead to other boxes that do not have lines going through the circle...

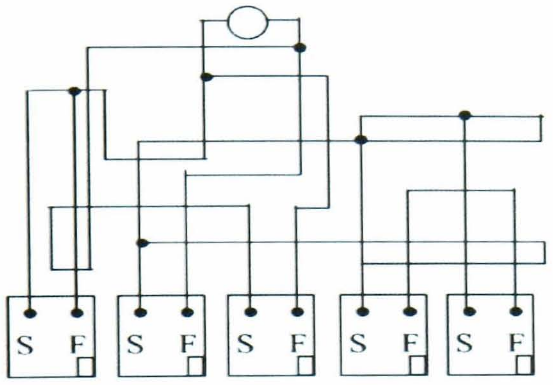
.... Two more practice examples are given here including the answers......

Your score on this test will be the number of problems marked correctly minus a fraction of the number marked incorrectly. Therefore, it will not be to your advantage to guess unless you are able to eliminate one or more of the answer choices as wrong....................... You will have 7 minutes for each of the two parts of this test. (Part 1 and Part 2) ....

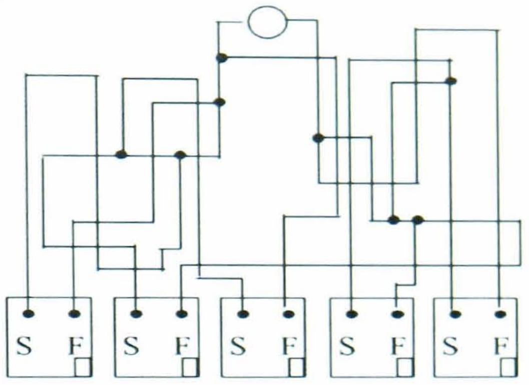
DO NOT TURN THIS PAGE UNTIL ASKED TO DO SO.

Part 1 (7 minutes)

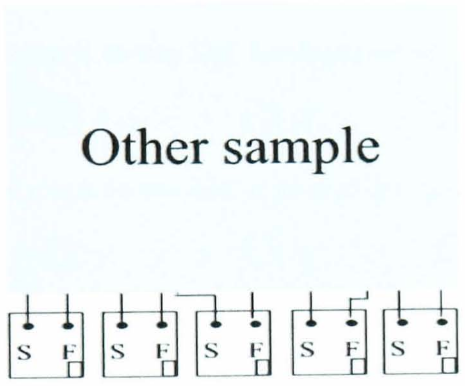
1.



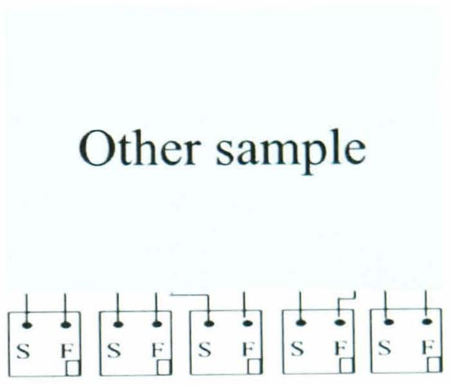
2.



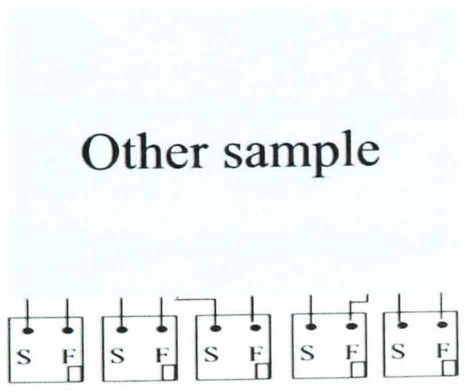
3.



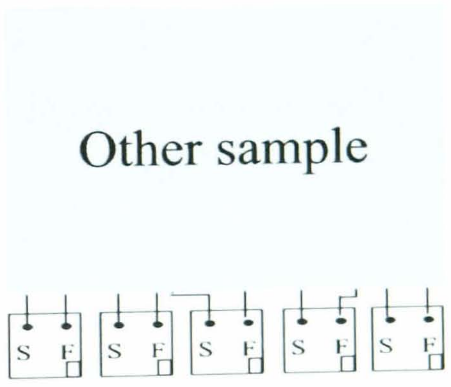
4.



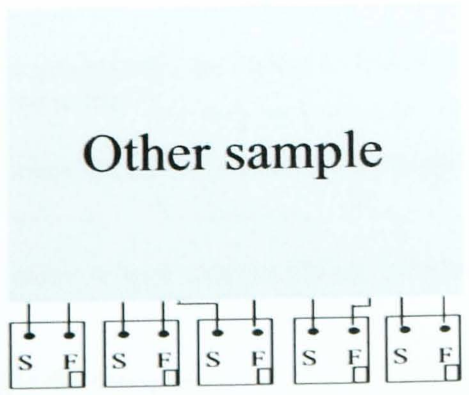
5.



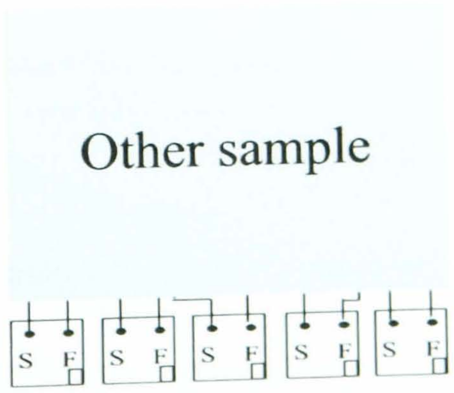
6.



7.



8.



2. Post-hoc questionnaire

Questionnaire

Thank you very much for your participation in my experiment a few weeks ago. We have got great results owing to your efforts!

All the information you enter will be **entirely confidential** and will be used for this research only. You can skip any questions if you do not wish to disclose the information but I will appreciate your answers very much.

1 Your department____________________________

2 Is this your first year at Brunel?______________________

3 Your gender: ☐ Female ☐ Male

4 Do you have a computer at home?______________________

5 On a scale 1 to 5 (1 being the worst and 5 being the best), please tick an answer for the following questions:

a. How good do you think you are at assembling home furniture such as book-shelves and tables?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

b. How much do you like drawings (any kind of drawings)?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

c. How much do you like or *used to* like playing with construction toys such as Lego?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

d. How much do you like playing computer games or Nintendo games?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

e. How good are you at getting to places in London using the London Underground?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

f. How much do you like games such as chess, puzzles, cross-words, naughts-and-crosses, etc?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

g. How good were you at programming (in any language) before entering Brunel?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

What are the languages?__________________________

h. How hard do you think the experiment was (1 being the easiest and 5 being the hardest)?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

6 Before you started your course ar Brunel have you used flowcharts or any flow diagram? What were they?____________________________

7 How many GCSE subjects did you achieve the following grades?

A*_______ A_______ B_______ C_______ D_______

8 How many A-level subjects did you achieve the following grades?

A_______ B_______ C_______ D_______ Below D_______

9 What is the newly adjusted mark you got in the experiment? (Ask Jarinee when handing this in, don't worry your identity will still be unknown.)______________

3. Textual program in the real test of Flow Study 2

```
Vegetables
If Careful then
        If Sad then
            Sausages
            Milk
        Else
            Bread
            Crisps
        End If
        Fish
Else
        Bread
        Milk
        If Funny then
            Cake
        Else
            Fish
        End If
End If
Eggs
If Picky then
        Chicken
Else
        Butter
End If
If Friendly then
        Jam
Else
        Salt
End If
Pay Bill
```

APPENDIX C

Prograph Evaluation

(Chapter 5)

C-1: Cognitive Dimensions of Notations Questionnaire

The Cognitive Dimensions questionnaire in the following pages have been taken from http://216.239.51.100/search?q=cache:6C4CKXLTcAEC:www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf+COGNITIVE+dimensions+questionnaire&hl=en&ie=UTF-8.

A Cognitive Dimensions Questionnaire

Alan Blackwell and Thomas Green

This questionnaire was developed as a tool for assessing the usability of information devices by means of the Cognitive Dimensions of Notations framework.

For further reading on the framework, see:
http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/

To download the current version of this questionnaire, see
http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf

We would be extremely grateful to be kept informed of the use of the questionnaire. If you contact us before sing it, we will be able to supply any recent amendments – both in format and analysis techniques.

ontact:

lan Blackwell	Thomas Green
lan.Blackwell@cl.cam.ac.uk	Thomas.Green@ndirect.co.uk
omputer Laboratory	Computer-Based Learning Unit
ambridge University	Leeds University

his version:
.1.0 November 2000

Thinking about Notational Systems

This questionnaire collects your views about how easy it is to use some kind of notational system. Our definition of "notational systems" includes many different ways of storing and using information – books, different ways of using pencil and paper, libraries or filing systems, software programs, computers, and smaller electronic devices. The questionnaire includes a series of questions that encourage you to think about the ways you need to use one particular notational system, and whether it helps you to do the things you need.

Section 1 - Background information:

What is the name of the system?	
How long have you been using it?	
Do you consider yourself proficient in its use?	
Have you used other similar systems? (If so, please name them)	

Section 2 - Definitions:

You might need to think carefully to answer the questions in the next sections, so we have provided some definitions and an example to get you started:

The ***product*** is the ultimate reason why you are using the notational system – what things happen as an end result, or what things will be produced as a result of using the notational system. This event or object is called the product. Any product that needs a notation to describe it usually has some complex structure.

The ***notation*** is how you communicate with the system – you provide information in some special format to describe the end result that you want, and the notation provides information that you can read. Notations have a structure that corresponds in some way to the structure of the product they describe. They also have parts (components, aspects etc.) that correspond in some way to parts of the product.

Notations can include text, pictures, diagrams, tables, special symbols or various combinations of these. Some systems include multiple notations. These might be quite similar to each other – for example when using a typewriter, the text that it produces is just letters and characters, while the notation on the keys that you press tells you exactly how to get the result you want. In other cases, a system might include some notations that are hard for humans to produce or to read. For example when you use a telephone the notation on the buttons is a simple arrangement of digits, but the noises you hear aren't so easy to interpret (different dialling tones for each number, clicks, and ringing tones). A telephone with a display therefore provides a further notation that is easier for the human user to understand.

Complex systems can include several specialised notations to help with a specific part of the job. Some of these might not normally be considered to be part of the system, for example when you stick a Post-It note on your computer screen to remind you what to write in a word processor document.

There are two kinds of these sub-devices.

- The Post-It note is an example of a ***helper device***. Another example is when you make notes of telephone numbers on the back of an envelope: the complete system is the telephone *plus* the paper notes – if you didn't have some kind of helper device like the envelope, the telephone would be much less useful.
- A ***redefinition device*** changes the main notation in some way – such as defining a keyboard shortcut, a quick-dial code on a telephone, or a macro function. The redefinition device allows you to define these shortcuts, redefine them, delete them and so on.

Note that both helper devices and redefinition devices need their own notations that are separate from the main notation of the system. We therefore ask you to consider them separately in the rest of this questionnaire.

To review how we intend to use these terms, consider the example of typing business letters on a word processor. The *product* of using the word processor is the printed letter on paper. The *notation* is the way that the letter looks on the screen – on modern word processors it looks pretty similar to what gets printed out, but this wasn't always the case. If you want to find and replace a particular word throughout a document, you can call up a *helper device*, the search and replace function, usually with its own window. This window has its own special notation – the way that you have to write the text to be found and replaced, as well as buttons that you can click on to find whole words, or to find the word in upper and lower case etc.

Section 3 – Parts of your system:

What task or activity do you use the system for?	
What is the *product* of using the system?	
What is the *main notation* of the system?	

When using the system, what proportion of your time (as a rough percentage) do you spend:	
Searching for information within the notation	%
Translating substantial amounts of information from some other source into the system	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new ideas in the notation, without being sure what will result	%

ɅAre there any *helper devices*? ʾlease list them here, and fill out a ;eparate copy of section 5 for each ›ne.	
re there any *redefinition* :vices? ease list them here, and fill out a parate copy of section 5 for each one.	

Section 4 – Questions about the main notation:

How easy is it to see or find the various parts of the notation while it is being created or changed? Why? What kind of things are more difficult to see or find? If you need to compare or combine different parts, can you see them at the same time? If not, why not?	
When you need to make changes to previous work, how easy is it to make the change? Why? Are there particular changes that are more difficult or especially difficult to make? Which ones?	
Does the notation a) let you say what you want reasonably briefly, or b) is it long-winded? Why? What sorts of things take more space to describe?	
What kind of things require the most mental effort with this notation? Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?	
Do some kinds of mistake seem particularly common or easy to make? Which ones? Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?	
How closely related is the notation to the result that you are describing? Why? (Note that in a sub-device, the result may be part of another notation, rather than the end product). Which parts seem to be a particularly strange way of doing or describing something?	

When reading the notation, is it easy to tell what each part is for in the overall scheme? Why?

Are there some parts that are particularly difficult to interpret? Which ones?

Are there parts that you really don't know what they mean, but you put them in just because it's always been that way? What are they?

If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden?

In what ways can it get worse when you are creating a particularly large description?

Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?

How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not?

Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?

Can you try out partially-completed versions of the product? If not, why not?

Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this?

What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?

If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Please give examples.

Are there places where some things ought to be similar, but the notation makes them different? What are they?

Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation?

If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw?

Do you ever add extra marks (or colours or format choices) to clarify, emphasise or repeat what is there already? [If yes: does this constitute a helper device? If so, please fill in one of the section 5 sheets describing it]

Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they?

Does the system insist that you start by defining new terms before you can do anything else? What sort of things?

If you wrote here, you have a redefinition device: please fill in one of the section 5 sheets describing it.

Do you find yourself using this notation in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?

After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they?

Could it be improved specifically for your own requirements?

Section 5 – Questions about sub-devices:

Please fill out a copy of this page for each sub-device in the system.

This page is describing (tick one box): a helper device ☐, or a redefinition device ☐

What is its name?	
What kind of notation is used in this sub-device?	

When using **this sub-device**, what proportion of the time using it (as a rough percentage) do you spend:	
Searching for information	%
Translating substantial amounts of information from some other source into the system	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new ideas in the notation, without being sure what will result	%

In what ways is the notation in this sub-device different from the main notation?
Please tick boxes where there are differences, and write a few words explaining the difference.

[VIJU]	Is it easy to see different parts?		
[VISC]	Is it easy to make changes?		
[DIFF]	Is the notation succinct or long-winded?		
[HMOS]	Do some things require hard mental effort?		
[ERRP]	Is it easy to make errors or slips?		
[CLOS]	Is the notation closely related to the result?		
[ROLE]	Is it easy to tell what each part is for?		
[HIDD]	Are dependencies visible?		
[PROG]	Is it easy to stop and check your work so far?		
[PROV]	Is it possible to sketch things out?		
[PREM]	Can you work in any order you like?		
[CONS]	Are any similarities between different parts clear?		
[illegible]	Can you make informal notes to yourself?		
[illegible]	Can you define new terms or features?		
[illegible]	Do you use this notation in unusual ways?		
	How could the design of the system be improved?		

Section 5 – Questions about sub-devices:

Please fill out a copy of this page for each sub-device in the system.

This page is describing (tick one box): a helper device ☐, or a redefinition device ☐

What is its name?	
What kind of notation is used in this sub-device?	

When using **this sub-device**, what proportion of the time using it (as a rough percentage) do you spend:	
Searching for information	%
Translating substantial amounts of information from some other source into the system	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new ideas in the notation, without being sure what will result	%

In what ways is the notation in this sub-device different from the main notation?
Please tick boxes where there are differences, and write a few words explaining the difference.

[VIJU]	Is it easy to see different parts?		
[VISC]	Is it easy to make changes?		
[DIFF]	Is the notation succinct or long-winded?		
[HMOS]	Do some things require hard mental effort?		
[ERRP]	Is it easy to make errors or slips?		
[CLOS]	Is the notation closely related to the result?		
[ROLE]	Is it easy to tell what each part is for?		
[HIDD]	Are dependencies visible?		
[illegible]	Is it easy to stop and check your work so far?		
[illegible]	Is it possible to sketch things out?		
[illegible]	Can you work in any order you like?		
	Are any similarities between different parts clear?		
	Can you make informal notes to yourself?		
	Can you define new terms or features?		
	Do you use this notation in unusual ways?		
	How could the design of the system be improved?		

Section 5 – Questions about sub-devices:

Please fill out a copy of this page for each sub-device in the system.

This page is describing (tick one box): a helper device ☐, or a redefinition device ☐

What is its name?	
What kind of notation is used in this sub-device?	

When using **this sub-device**, what proportion of the time using it (as a rough percentage) do you spend:	
Searching for information	%
Translating substantial amounts of information from some other source into the system	%
Adding small bits of information to a description that you have previously created	%
Reorganising and restructuring descriptions that you have previously created	%
Playing around with new ideas in the notation, without being sure what will result	%

In what ways is the notation in this sub-device different from the main notation?
Please tick boxes where there are differences, and write a few words explaining the difference.

[VIJU]	Is it easy to see different parts?		
[VISC]	Is it easy to make changes?		
[DIFF]	Is the notation succinct or long-winded?		
[HMOS]	Do some things require hard mental effort?		
[ERRP]	Is it easy to make errors or slips?		
[CLOS]	Is the notation closely related to the result?		
[ROLE]	Is it easy to tell what each part is for?		
[HIDD]	Are dependencies visible?		
[PROG]	Is it easy to stop and check your work so far?		
[illegible]	Is it possible to sketch things out?		
[illegible]	Can you work in any order you like?		
[illegible]	Are any similarities between different parts clear?		
	Can you make informal notes to yourself?		
	Can you define new terms or features?		
	Do you use this notation in unusual ways?		
	How could the design of the system be improved?		

C-2: Immersion Diary

Content from the diary obtained during the immersion into learning Prograph, a list of problem tokens is generated and organised into a tabular fashion below. Column 'Description' contains the scripts copied from the diary without any modification. A script can have more than one problem associated with it. At the end of this table there are 21 figures which are referenced by the content in the table.

Problem tokens from the diary

Problem tokens are numbered in chronological order. The number in the bracket refers to the number of sub-problems (issues) it is further divided into.

Problem	Description
1 (3)	When there are many windows on the screen, only the active window has text description of the window on the title bar. This makes it difficult for learners. I often gets lost, wondering where I am in..particularly when the active window is down the hierarchy . However, this seems to be an available feature because in the tutorial, a greyed title bar with visible text description is seen.
2	The 'get' operation...the left root is not linked to anything else (only in this particular case), so why is it there. OK, it is supposed to mean that the instance is obtained and passed through the get operation, but this is not obvious.
3	The 'set' operation...the root hangs there without data link (see figure below), why hanging there? This could cause confusion. It is also logically inconsistent.
4	How intuitive or representative to the meanings are the icons?
5	To start a new project, I messed around a bit not knowing how to. I had to go to File then Close Project. Once that's done, the Untitled sections window showed up. This should be easily achieved by just clicking a New icon which automatically closes the working project.
6	A new window always stay on top of the old window. It would be better if it is placed next to the old window if there is space available.
7	'Get' and 'Set' operation icons are sometimes confusing...which is 'get' and which is 'set'? Perhaps a G and a S somewhere in the icons can be a good reminder. *(see Figure 1)*
8 (2)	The separation between Class attributes and Instance attributes with a green line is not obvious. Suggestions are: 1) make the icons look distinctly different. 2) put the green line a bit below that so that the upper section is wider and noticeable but perhaps this is more error prone, so maybe the two sections should be in different colours or give some sort of indication.
9	It seems that the left root of an operation is default to the flow of the instance and that the right root is for a value/data from that instance. However, how can we know it? Error occurs when the left root is linked to an Evaluation operation because of data type mismatch. Maybe the left root should be a different shape or different coloured to indicate that it is meant to be the flow of an instance.

Problem tokens from the diary (cont'd)

Problem	Description
10	BIG IDEA. - get art/design students to design icon. The icons must reflect the intended meanings, must have space for naming, must be small enough to fit in a small window. - Get a group of programming students to evaluate these icons. See if the icons are classified or seen by the programming students as what they are intended for.
11	List as constant..., the items are separated by a space. A comma may be more intuitive...
12	The operation '(in)', must have brackets, I find it awkward.
13	The (in) operation returns value 0 if the item checked is not in the list. Is this intuitive? Can't it be 'not in list' or 'fail' instead of a zero.
14	The operation '"join"'..Join must be in double quotes. Awkward.
15	How do you pass back control to another case?
16	Can't undo!
17	Cases1,2,3,4....The numbers are not meaningful. What would be nice is if a short descriptive name can be given as 'tool tip'.
18	When a 'local' is made by drawing a marquee around the operations, if an operation gets omitted accidentally, at present the only way to do is to 'cut' it, open the 'local', and paste it in the 'local'. It would be nice if it can just be dragged and dumped on the newly created 'local'.
19	Would be nice to have an icon for executing method.
20	Error messages in the bottom bar are rather difficult to understand. Oh, I no, the old message stays there although it is not valid any more. This confused me at first until I noticed that the Prograph tab at the bottom of the screen blinked together with the new message. I think the old message should either not be there or change in colour to say that's the old one.
21	The concept of the 'fail control' is new to experienced programmers...so how does it fare to novices. Probably same? [Finish, Terminate, Fail]. I found it really difficult to set 'fail on success', 'fail on failure' in different windows as in the last example exercise. This could be a very difficult concept to deal with for novices. I survived though (by using the step into facility).
22	Why is the Beep operation start with an upper case while other primitive operations such as 'show' and 'ask' start with a lower case?
23	I always double-click the method name to open the method window. But it doesn't. Double-clicking lets one rename the method name. To open the method window, one has to double-click the method icon!
24	We want a facility to arrange icons in universal methods window, in particular. Windows/Tidy and resize does the job but not perfectly. Windows/Arrange icons does not do anything at all..why?

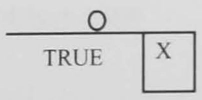
Problem tokens from the diary (cont'd)

Problem	Description
Pos-1	Windows/View by Name is good, it lists the methods.
25	In the message box below, the position of 'select' and 'cancel' should be swapped, I think. I mistakenly clicked 'cancel' when I wanted 'select'. *(see Figure 2)*
26	I had a hard time trying to figure out the Next Case below (I lost the picture!). It seems that with the x control if passed onto 'show' and with the tick, it goes to the next case. What's the logic behind?????
27	I think Program **iteration** may worth investigating: whether the concept is difficult.
Pos-2	'Evaluation' is a good idea for representing an equation requiring little space for it.
28	['Evaluation' is a good idea for representing an equation requiring little space for it.] However, the implementation is limited to using a, b, c... in that order for the input from left to right. So there is no freedom for the programmer to use x or y if he wants to. To do that, it is very clumsy and messy, ...
Pos-3	Prograph lets you call methods before creating them and also lets you create them on a fly, while the program is executed. The process is called prototyping. It encourages top-down step-wise programming. This is actually similar to my model.
29	When trying to create another terminal and if it is too close to the existing one, Prograph gives an error message that it's too close. Why doesn't it just stretch the icon automatically and add a terminal without giving the error message? It is a nuisance. Actually, Prograph does do it for you automatically but only when you click far enough and if the new location is the last one on the right or on the left. Therefore, <u>the minimum required distance (threshold) between the existing terminal and the new location seems to be too large</u>. Either set it much lower or don't have it at all since if user does intend to create a new terminal any way (by double-clicking when the cursor changes into cross-hair).
30	The 'not equal to' sign is ~=. Is this intuitive?
31 (2)	Couldn't find the feature that will END the program in the middle of everything else like in VB.
32	Can there be two match operations leading to two next cases? I had two match operations, 2 next cases, a total of 3 case windows. But Prograph always open the second case window for both matches. Prograph should have a better way of implementing 'Do Case'. I want to be able to say 'go to case 2' or 'go to case 4', not just to say 'go to the next case' when there are more than two cases!
33	What is Prograph equivalence of Function?
34	Check p 78 on 'Control Construct' in the text Prograph. *(see Figure 3)*
35	The operation join has to be typed as "join" with double quotes. This is a source of error because other primitive operations such as show and ask do not require it. Although the double quotes are there for a purpose: to remind you that this operation only concatinate strings. If the programmer has to learn the conept of concatenation anyway, the name of this operation might as well be *concatenate*. Otherwise, *append* may be used as Prograph seems to impose order of putting the items together by the ordinal position of the terminals on the node. Or else, find out what word is the best..would join be the best? Wouldn't this confuse with relational database operation , *join*?

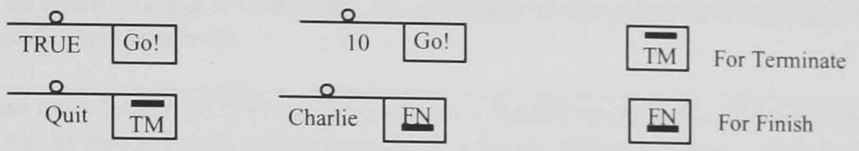
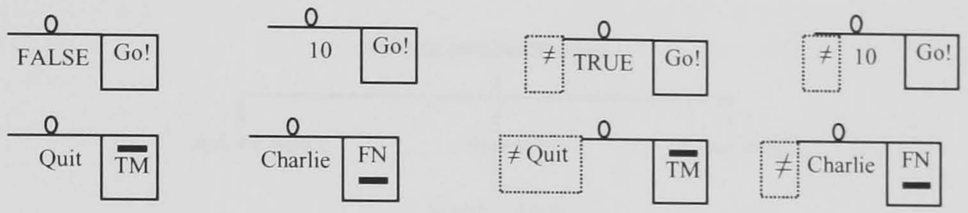
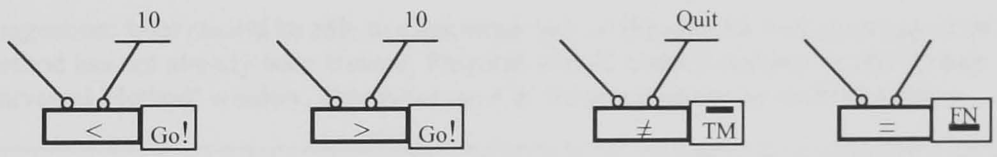
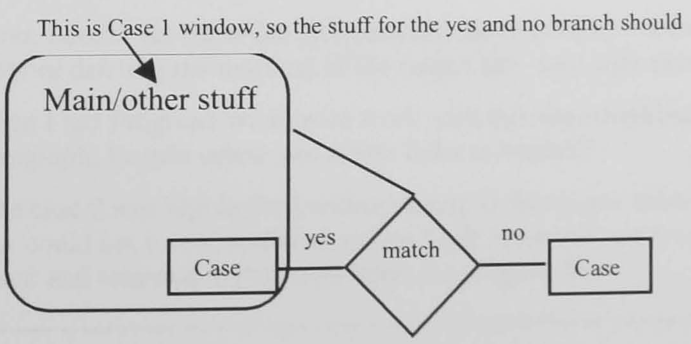
Problem tokens from the diary (cont'd)

Problem	Description
36	Prograph implements *string* differently from other conventional languages, which treat a *string* as an array of characters. In Prograph then, how can one manipulate strings such as printing a phrase/word in reverse order? Is it by converting the string into a list of character first????
Pos-4	*Inject* is a good idea to me, but is it, to others?
37 (2)	I had a problem figuring out how to resize an operation and needed to consult Help. I could figure out how to make the operation bigger by dragging a terminal outward, but to make it smaller when there are many terminals on the node is not easy. After the Help, I could do it but still had confusion on the order of the terminals on the show operation. The order must be correct to display the message correctly. So a trouble occurred when all terminals clutter towards one side (not balance) and hence dragging the most outer terminal towards the centre accidentally swapped its position with the next one (because they were too close), causing the message displayed in a wrong order.
Pos-5	I found programming is very easy because I could just create dummy methods (like dummy procedures) and filled in the code later.
Pos-6	There is very little typing. But the same is true for TPL. The only advantage is that there is <u>less typing</u> and methods can be created/coded on a fly unlike in TPL where you will get an error message. However, this is a special feature provided by Prograph. I suppose TPLs could also let programmers write the procedure on a fly!
38	The loop! It requires that there is a data item output from each iteration back to the loop. Otherwise, there is no data input for the next iteration. This is natural. Nevertheless, I often forgot about it and got an error message. Prograph automatically puts a terminal on the output bar of all case windows for the method as a default. But this is still not enough. May be this is because I am not used to it yet. To me if I see a root, I know I need some output, but this does not look like a root so it slips my mind. I have no suggestion for this!
Typo-not a usability problem	Mis-spelling in Prograph error message (Typo: T-2): "The inputs in this primitives cannot be compared <u>beacuse</u> they have incomparable types."
Typo-not a usability problem	Mis-spelling, in the Help, Index, under loop, 'lopp annotations ..icons' (Typo: T-1).
39 (2)	Couldn't find a short cut key to abort when get into an infinite loop. The keys given in the text (p.92) didn't work!
40 (2)	Passing control when dealing with loops is very difficult. I didn't know that you can put a terminate control next to a multiplex to stop the outer loop. See the section 'test for +step'. I still don't quite understand why it worked eventually.
41 (2)	The 'finish' and 'terminate' controls enforces 'lookahead' (to avoid premature commitment).

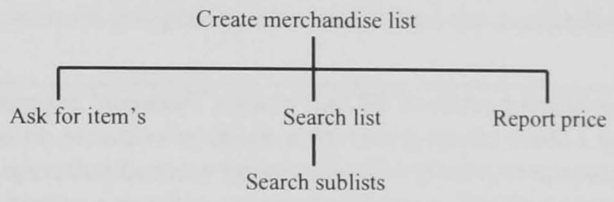
Problem tokens from the diary (cont'd)

Problem	Description
42	I am still struggling with loops!
Typo-not a usability problem	Mis-spelling in the HELP--Extensions Reference.
43 (2)	**'accept** Description Opens a Value window with, optionally, titleanainitial alue.' When I tried to use 'accept' it gave the following msg: "This external was not found in the .DLL 'Primitives'. Make sure the External Definition file containing this external was created properly." This msg is incomprehensible.
44	The Help would be better with a screenshot of what the primitives do. For example when using 'answer', 'select, 'answer-v', ...what do the user see? For example, the following is for 'select' *(see Figure 4a).* The code is *(see Figure 4b).*
45 (2)	The primitive 'ask' is probably better called 'prompt'. And the 'ask' should be able to do what can be achieved by only the 'show' used together with the 'ask', such as, prompting the user with a combination of string constants and number values (data items) and at the same time getting a response/answer from the user. *(see Figure 5)*
46	Always have to bundle up a group of operations into a 'local' when I want to repeat that. But if there are only a few operations, it would be nice not to hide them in a local.
47 (2)	It would be nice if numbers can be defaulted to strings by the system like in VB, depending on context. For example, when "join" strings. Strings are required but number should be automatically changed to a string by the system.
48	TERMINATE/FINISH.... When in the second case, if we want to terminate after it finishes execution without doing subsequent operations in the first case, just put a TERMINATE control next to the method or the local in the second case window.
49	IDIV primitive. There are two roots: quotient (left) and remainder (right+default). If we want to link the remainder to something else but do not want to use the quotient. We still have to put both roots in otherwise Prograph thinks that the link is for quotient.
50 (2)	Couldn't apply Partition Annotation, couldn't find Partition Menu command.
51 (2)	Hard Mental Operation!..The next case for the match test is hard. Always I have to think carefully if it should be the tick or the cross for what I want to do. And I make mistake very often even after doing it carefully. The mistake is only caught out by debugging during the execution. Particularly crazy is this: *(see Figure 6)*  This is confusing in the head! The TRUE and the $\boxed{X}$ are contradictory although they mean 'If Not True Then Go To Next Case'. It's hard and I had to do **double thinking (2-steps!)**.

Problem tokens from the diary (cont'd)

Problem	Description
	Here are some suggestions:  TRUE Go! 10 Go! TM For Terminate Quit TM Charlie FN FN For Finish What about the false case then?  FALSE Go! 10 Go! ≠ TRUE Go! ≠ 10 Go! Quit TM Charlie FN ≠ Quit TM ≠ Charlie FN The above are examples for the 'Match' representation. Below are examples for when we use an operator. This is much easier because it is straightforward. People will usually want to express only the true case. Who wants to think twice? Prograph allows the possibility of saying: if it is 'not less' than 10 then go to the next case. This is unnecessary! The above representations are actually not as good as the ones below and may prove unnecessary. The above representations are actually not as good as the ones below and may prove unnecessary.  10 &lt; Go! 10 &gt; Go! Quit ≠ TM = FN On another note, the original Prograph Tick and Cross may be OK if they are put in front of the value item, so it can be read like, if not greater than 10 go to next case. However, I tried but still didn't find it helpful. Perhaps this is because it is not visible that the stuff in the first window is for 'yes' case or is one of the branches from the IF or Case construct. (poor Visibility) I see the problem now! We mix a branch of the IF construct (Case construct) in the main (previous section) of program while separate out the other branch (other branches). Why not having something like this:  This is Case 1 window, so the stuff for the yes and no branch should Main/other stuff Case yes match no Case

Problem tokens from the diary (cont'd)

Problem	Description
52	Why is it that after the 2nd case finishes, it returns to the 1st case? And this awkward step has to be taken? To put a TERMINATE icon there at the end is very awkward and unnatural! *(see Figure 6)*
53	It would have been nice if Prograph provides a facility to show the hierarchy (tree structure) of method calls within a program. I found myself lost quite often and wanting to know who calls whom! An example of a tree structure is this:  Create merchandise list Ask for item's Search list Report price Search sublists This reminds me to question of Hidden Dependency. A tree structure like the one above would have the programmer to make sure he double-checks affected methods once one method is modified.
54 (2)	Prograph lets you create a *skeleton* universal method . It can be created on a fly while the program is running. This is fine, you are given the 'Create Universal Method' window to specify where you want it to be. However, if you decide that you want to do it while editing the program, you must click the left side of the icon to get that window. The **problem** is that I always click the right-hand side of the icon and get an error message (which, by the way, is hard to read as it is in red and is small! This is because Prograph expects that the right-side click will open an existing method and the left-side click will 'create' the method. Suggestion: User should be able to click either side of the icon for both purposes. If the method has not already been created, Prograph should know it and pop up the "Create Universal Method' window. Otherwise, as it is, Prograph opens the method window
Pos-7	The case window …1,2,3… shows title when cursor is on it. But only if the programmer remembers to title each case via commenting the input bar. It's good in terms of highlighting 'functional' information type.
55 (2)	A bug in Prograph! See below: *(see Figure 7)* Once the stuff inside case 2 was Cut, I returned to case 1, highlight case 2 icon, right-click and chose Delete. I tried to delete the link between the input and output bars. No Success!. I couldn't do anything with any of the links. Even when I pasted the stuff I Cut from the case 2 window and tried to create links from it to the bars, I had no success. However, I could get rid of the link between the two bars by deleting the root of the input bar or deleting the terminal of the output bar. And after that everything is OK. But when I left Prograph window to work with this document and then returned to work with Prograph, I could delete and create links as normal! Or if the case 2 was highlighted without being Deleted (see below), link in case 1 window could not be manipulated, unless I left Prograph window to work with this document and returned to Prograph later. *(see Figure 8)*

Problem tokens from the diary (cont'd)

Problem	Description
56 (3)	Prograph implementation of case (window) is too restrictive. See the example below. *(see Figure 9)* For programmer to do it intuitively a Tick (for TRUE) will be used, in which case, the program is longer and Visibility is reduced. In order to increase Visibility, a Cross (for FALSE) must be used, leading to Hard mental operation! On the other hand, if Branching is allowed within the same window, these problems will be resolved. Nevertheless, this leads to Inconsistency and perhaps it is the reason why Prograph does not allow this feature (as the Boxes and the Gates notations in LabVIEW). The question is whether it is worthwhile to allow both features. From LabVIEW experience, perhaps it is as there seems to be no complaints from users about the availability of the two features co-existing.
57	The primitive operation 'number?' returns FALSE even for a string consisting of numbers. There is no primitive to check if the string can be made a number. We need such a primitive operation because before using the primitive operation 'from-string' to change an string item to a number, we must make sure that the string does not have anything but 0 to 9.
58	Sometimes when there is not much code, it will be a lot more Visible if we can put all the code in the same window. See below: *(see Figure 10)*
59	Always forgetting to change the root and terminal to 'loop'!!!
60 (2)	Why not having an operation called "success" for this use (see below)? I found it confusing to choose between terminate and fail and also the tick and the cross! *(see Figure 11)*
61	HELP--unclear --see below <any*> what does the * mean? Well, I could guess but I could not be sure. from-string Description Returns the value textually represented by String. Type cannot be a class or External structure. Inputs String <string>: Outputs Data <any*> | Point | Rect | RGBType Note In producing output, this primitive follows Prograph rules for unparsing. For details on data types, refer to the Prograph CPX User Guide See also from-ascii, to-ascii, to-string, tokenize
62 (3)	Again! I have problem with match test. The tick and the cross are just not natural to me. When I want to say that 'if it is false then terminate', I should give value FALSE, a tick and a terminate. Instead of doing that, I gave a FALSE, a cross (by default), and a terminate. This event occurs very often with me. I only realised after executing the program. This is because I have to think twice (as mentioned before). This is Hard Mental Operation!!!! Suggestion: should have only a Tick, not a Tick or a Cross.
63	How can I do the equivalent of this in Prograph? See below. X="123a" If not number(x) then Write("It's not a number!") End If Write(X) Here I want the program to write both 123a and it's not a number if it is not a number! However, I can't find the way to do it without repeating some of the code in two case windows. I am aware that the problem may be because the textual language is a control flow one and Prograph is a data flow language. The Prograph code below will only write either of the two messages. How can I say do this and this and then go to case nth? This is a Control of Flow problem. *(see Figure 12)*

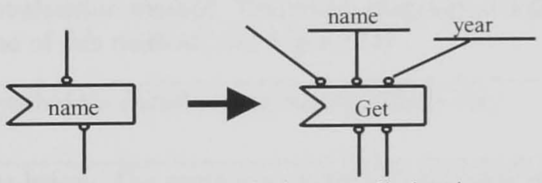
Problem tokens from the diary (cont'd)

Problem	Description
64	HELP--information is missing. Could not find information on "in" at first because I didn't know that it is listed under 'prefix primitive' in the HELP.
65	Imposed Look-Ahead The primitives such as prefix primitives require that the user know the order of their implementation. For example, we should know that the "in" primitive will start checking the input string from left (1st position).
66	**Starting OOP** The line separating class attributes and instance attributes is a bit too high when no class attribute. It is not noticeable. Suggestion: separating the two regions with a thicker line, or colour the regions differently, or labelling them.
67 (4)	The 'class method window' and the 'universal method window' are confusing for novices. Both windows are not normally open side by side. When only the 'class window' is seen, it is not easy to know which 'method' icon it came from. Suggestion: what about either adding a little 'class' icon next to the 'method' icon' or replace the word 'car' with 'Class method: Car'? *(see Figure 13)*
68 (2)	How representative are the icons for instance, class, attributes, method, section, and persistent?
69 (3)	Should the class method and the universal method have same or different icons? O-Oh! they are actually different. See the pictures above. The icon representing class method, Car, is 2-dimensional whereas the one for universals of Car is 3-D. This says, the difference is hardly noticeable, at least not by me after about two weeks of Prograph. Suggestion: a. Modify the two icons, make them distinct from each other. b. Test users on 2 and 3.
Pos-8	Ability to add comment any where and move it or hide it ....good point.
70 (2)	What is a primitive for simple 'assignment'? There is only the 'set' operation to set attribute values but not for variables because *there is no concept of variable in data flow programming!* Maybe I look for it because I am influenced by my control flow experience.
71	How intuitive the '!' in this primitive is: 'set-nth!' !!!!
Pos-9	Easy (context-sensitive) access to HELP by clicking the RHS of the icon for the primitive operation. This comes in very handy.
72	Would be nice if Prograph HELP has a list of available primitive operations. There must be...find that out!
73	The biggest problem in Prograph is the Control-of-Flow problem. Below copied from Prograph User-Guide...Are they easy? Types of controls Controls on operations dictate an action to be taken on a particular condition and provide control flow in Prograph. The types of controls available are: • The Next Case control • The Terminate control • The Finish control • The Continue control • The Inject control • The Fail control.

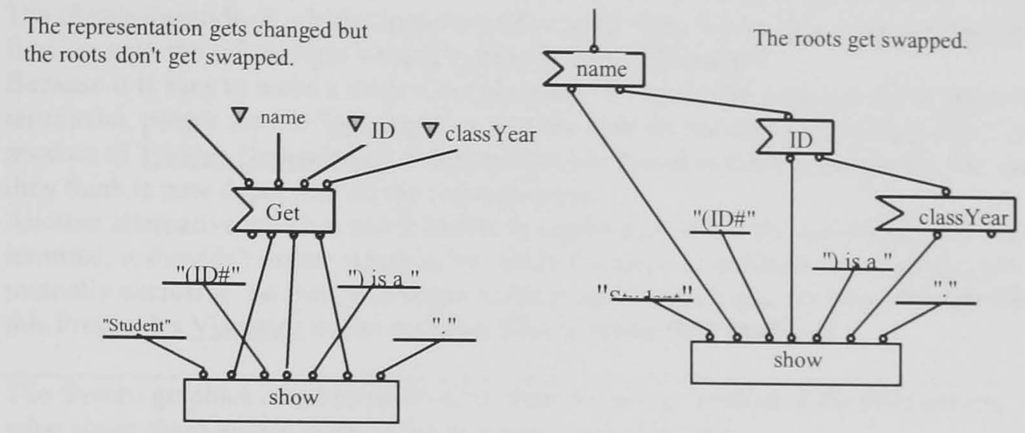
Problem tokens from the diary (cont'd)

Problem	Description
74 (2)	When the superclass method call icon is created, if a name is not given right there and then, a name cannot be given later and the icon will have a name '//' instead of '//abcd'. However, if a name has been given while the icon being created, it can be edited later…no problem.
75 (3)	I want a Case construct, the way used in procedural languages, not the Prograph's Case control: 1,2,3,4 windows! I want to be able to do a match test and say case 1, if =; case 2 if <; case 3 if >. Due to the lack of this feature, its Visibility is poorer than TPLs in this respect.
76	After highlighting and deleting case 2, I couldn't create links in case 1 window. I then activated a different Prograph window area and came back to it. Only then could I create a link!
77 (2)	Clicking the left side of a class method gets the error below while clicking the right side opens up the class method! *(see Figure 14)*
78 (2)	Subclass can't use method of parent class. The error message is: 'Input 1 is not compatible with the required type.' The parent class is Array, the subclass is Integer_array. The method in the parent class expects an array. But why shouldn't it be ok?
79	Error-Proness: In the order below from 1 to 3. When changing 'get array element' to a short name and then press Return, the link did not change but when changing from the short name to the long name (get array element) and press Return, the link wen to the left root. Either pressing Return or clicking the area outside gave the same result, i.e. the link changed! This is an automatic feature which is harmful! *(see Figure 15)*
80 (9)	So many windows!!!They are in a mess. As I worked along, the number of windows kept increasing, particularly when I tried to debug or understand the program. There is no one window that will give a big picture of the program. When working with objects, classes, inheritance, polymorphism, occasionally, I needed to see the 'class method' windows (both parent and children) quite often because I couldn't remember whether the method I wanted to use at the time was in the parent class or the child class. OK, I could go to the Window Menu and click for the windows I need, but I think it would be nice to reserve an area on one side of the screen for easy access to whichever windows are essential. For example, if working with classes, i.e., once sub class, super class has been defined, perhaps, have a tree structure of superclass and subclass methods on LHS like in the Windows Explorer, where one can just click the name of the class/subclass method to open its window. See Green and Petre(1996) p. 155. It said Prograph had a large number of long-range hidden dependencies, that it ought to provide a facility to search the ancestors of a given method, and that there was a searching tool for this purpose provided. But what is the tool? I think the tree structure I suggested above should do the trick. At the moment, programmers can search from parent methods (up in the hierarchy) to children methods (down below in the hierarchy) but not the other way around.

Problem tokens from the diary (cont'd)

Problem	Description
81 (2)	There are 4 ways to name operations that reference methods: *explicit-class method reference; context-determined method reference; data-determined method reference; and universal method reference.* This is quite confusing for beginners. And it is very hard to implement when the screen is in such a big mess! So far the BIG problems are - Control Flow (Hard mental operation, Role-Expressiveness (see Green and Petre, 1996, p. 158), Enforced lookahead) - Inheritance and Polymorphism (Visibility-need a clear representation of its structure) - Method Reference (Hard mental operation).
82	When in Windows/View by Name mode, Prograph automatically re-arranges the icons in method windows alphabetically and immediately right after the carriage-return key is hit once the new method name is entered or edited. While this seems a good facility to have, I often found it a potential source of (slight) delay and error. This was because I didn't notice the newly created/edited icon had been moved to another location and would double-click the last one (where I was) to open it. a. If I hadn't notice that the code in the newly-opened window is different from what I expected, I might have gone ahead modifying the code in the wrong method.--Error-Prone! b. If I noticed it, I would have to close the window and find the right icon in the method window before I could modify the code.--Delay! When it is not in the View by Name mode, icons are left where they were created...rather messy!
83 (2)	The operation Get: It gets the value of an attribute (of an instance of object), one at a time. If we need to get a few of them, the screen would become clutterred. It would be nice to be able to specify more than one attribute to get by the Get operation. Suggestion:  Benefit: The Get and Set icons are not obvious or distinctive anyway. By specify the name of the operation as suggested, this problem is mitigated. Another benefit is that the new representation allows more attribute names to be specified, hence less clusterred screen results.
84	If we are going to keep the existing Get and Set operations, what about swapping the two roots? The benefit will come when Show is to be used to show all the attribute values. *(see Figure 16)* Look at the two alternative representations of the code on the LHS given by the drawings below. The grey triangular shape represents an attribute (Prograph's representation for attribute). The right representation looks better than the left one. The representation for Get used by Prograph is Inconsistent with the convention used for primitive operations, i.e. putting the name of the operation within the box. For the Get and Set operations, the icons themselves have individual meaning. Get and Set are, to the programmer, 'operations' as well as 'show', 'ask', etc. Why do they have different iconic shapes, and, besides, have got no name as if the shapes themselves told the readers what they mean so clearly?

Problem tokens from the diary (cont'd)

Problem	Description
	 The representation gets changed but the roots don't get swapped. name ID classYear Get "(ID#" ") is a " "Student" "." show The roots get swapped. name ID classYear "(ID#" ") is a " "Student" "." show
Pos-10	List processing capabilities are Prograph's strong point.
85	Prograph does not provide Array data type but has List instead. Nevertheless, sometimes array is needed and it could become troublesome to impose on the programmer to implement the array data type by himself, even though that is possible. The question is whether we should have: array; list; or both.
86 (2)	When calling a class method (by prefixing the name with a slash), clicking the right side of the rectangle open the code window, but clicking the left side gives error message: there is no universal method….want to create? We should be able to click either side in this case. This is confusing *(see Figure 17)*.
87 (2)	"An Initialization method is always given the name <<>> by the Prograph editor."(..from HELP) So why does Prograph allow me to edit a name in the <<>>? The program worked even if I mistyped the name of the initialisation method. This means that its name is not taken into account at all as the method is contained with the class and there can be only one initialisation method. Therefore, Prograph should disable the editing facility for the name of this method. *(see Figure 18)*
Pos-11	Good thing! The symbol for initialisation method stands out.
88 (4)	Look at the pictures below. The same icon is used for all types of methods, which is understandable. We don't want to have to memorise so many icons. Prograph uses text to help programmers identify the method type. The first one, universal method window is not clear, comparing to the class method (class name/method name) or the initialisation method (the symbol says it all!). *(see Figure 19)* *(see Figure 20)* Observe these three layouts. Keeping the Prograph terminal arrangement (left for instance, right for value) and be 'neat', one has to use either A or C. Layout A and B impose no ordering, which is the virtue of data flow programming. Layout A is neat but maybe unnatural for right-handed people, who may be more comfortable with working from left to right. Layout C is probably easier for right-handed people but it imposes ordering. To swap or not to swap the two terminals: C is not affected. B will improve. A will be messier.

Problem tokens from the diary (cont'd)

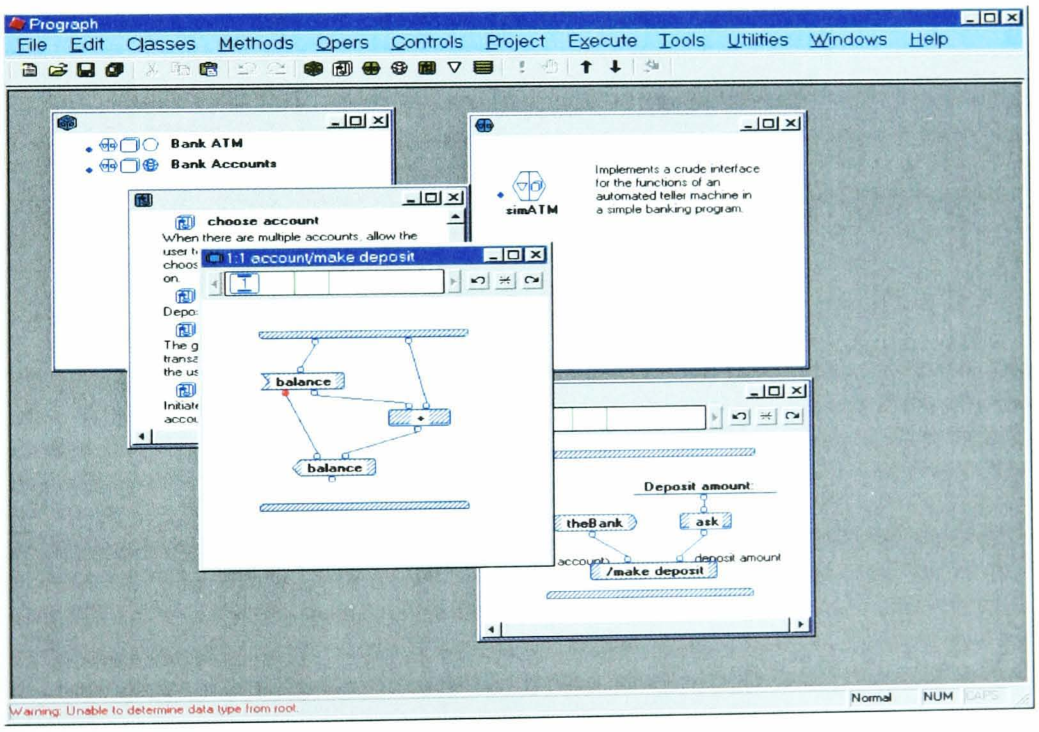
Problem	Description
	The choice depends on whether people tend to work from left to right, even among those familiar with the left-to right writing system or those left-handed. Because it is easy to make a mess if people prefer B (due to the arrangement of the two terminals), people start to 'lookahead' to tidy the code up and therefore end up with C, a product of 'Hidden Dependency'. Programmers are forced to think sequentially. The way they think is now dependent on the representation. Another alternative which is much harder to implement, is that for operations with two terminal, it shouldn't matter which is for which (instance or attribute value) as they are mutually exclusive. So then, it is down to the programmers' style. By being flexible like this Prograph's Viscosity is also reduced. This is **syntactical problem!**
89	The Syncro graphics could be made a bit more revealing. Instead of the little curves, what about sharp angles to make the direction more obvious?
Pos-12	Good point Prograph provides a facility to tidy links (straighten links and straighten linked node). This makes the diagram a lot less messy and easier to follow.
90	HELP--incorrect information. The information for the primitive **ask** gives two incorrect pieces of information: a) that there are Cancel and OK buttons but in actual fact there is only OK button; b) it references **accept** but I could not use the primitive! **ask** Description Opens a modal dialog prompting a user for input. The dialog has two buttons (Cancel and OK), an editable area, a textual prompt, and a default value in the editable area.. Canceled? <boolean> : True if the user pressed the Cancel button, false if they pressed the OK button. See also accept, answer, answer-v, select
91	The stuff in the HELP-User Guide is different from what is actually available. For examples, it shows the 'partition' option in the Control menus but in my copy of Prograph (2000) there is no 'partition' option. Another example is the options available in the Edit menu. The list in the User Guide is different from what I have.
92	That in the dataflow model there is no sequence. But when writing an application, I wanted to ask data from the user in a particular order. Though the program was written with my instinct of ordering (data flow being top to bottom made me subconsciously believe that there was an order the way). Although Prograph provides the 'synchro' for programmers to impose the order of execution, I tended to forget. Doesn't this impose a cognitive demand on programmers and this is inherent in the dataflow model itself, not the programming language.
93	I had trouble working out how to use Persistent to my advantage. The only sources I had were Prograph Tutorial examples and the text. Yet, they were not enough. I had to work it out.
94	**Prograph's Cases** ---Imposes Lookahead The following code can never be written because both next cases are always case 2. Prograph's way is rather clumsy for implementation of Do-Case as we cannot have two Next Case controls in the same window. *(see Figure 21)*

Problem tokens from the diary (cont'd)

Problem	Description
95 (3)	**Inheritance---<u>Viscous!!!!!!</u>** Parent class has to be created and its attributes filled in before child class is created. Otherwise, only the attributes but not their values will be shown in the child class attribute window. Worse still, once I implemented the program creating children classes before creating the links. Prograph allowed the links to be created later and showed the parent class attributes in the child class window but the program just wouldn't run because it did not recognise that an instance of the (supposedly) child class is the same class as the parent class. Even when everything worked fine (i.e., created in the correct order), changing the value and attribute name in a parent class attribute only changed the attribute name in the child class automatically, the value didn't change with it.

Figures referenced

<u>Figure 1.</u>



When there are multiple accounts, allow the user t choos on.

Implements a crude interface for the functions of an automated teller machine in a simple banking program.

Warning: Unable to determine data type from root.

<u>Figure 2</u>

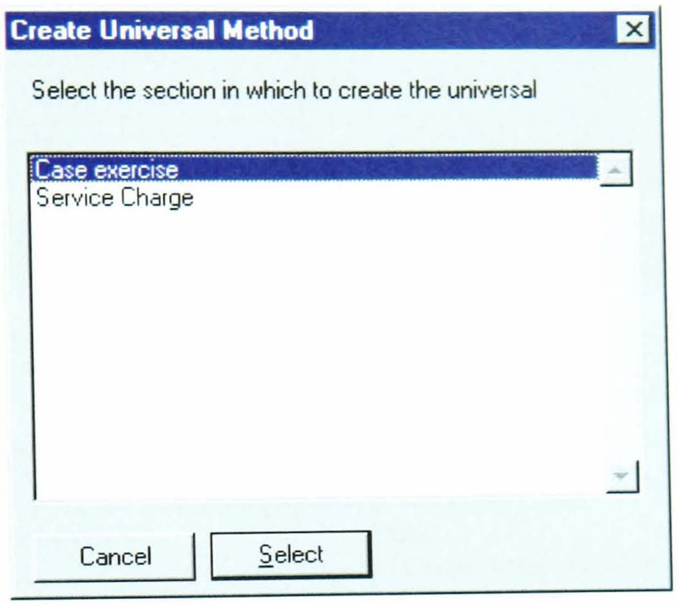

Create Universal Method
Select the section in which to create the universal
Case exercise
Service Charge
Cancel
Select

Figure 3

Selection Control Construct	Comment
1. Next Case on Failure '..*go to the next case immediately if the match test fails.*'	If Not...... Then Go to next case Else Do the code in this case End
2. Next Case on Success '*go to the next case immediately if the match test succeeds.*'	If......Then Go to next case Else Do the code in this case End
c. Continue 'The *continue* allows the remainder of the code in the current case to continue executing even if the match test fails. We use only *continue on failure* notation since *continue on success* would be an unneccessary test.'	What's the use? The code in the window gets executed anyway, either success or failure. What's the point of matching a value and don't care about the outcome from matching?
d. Terminate 'if the match conditions are met, the *terminate* control ensures that the current case is exited immediately and that its remaining code is not executed....This is especially useful for *..loops* and *repeats*, ..Not only does terminate stop the current execution of the method, it also prevents further *repetition* of the method.'	Isn't this condition for a Do_While?
e. Finish '..is similar to terminate. However, ...the finish control allows the remaining code of the current case to be executed once, but it prevents further repetition of the method in loops and repeats.'	Isn't this condition for a Repeat_Until?

Figure 4

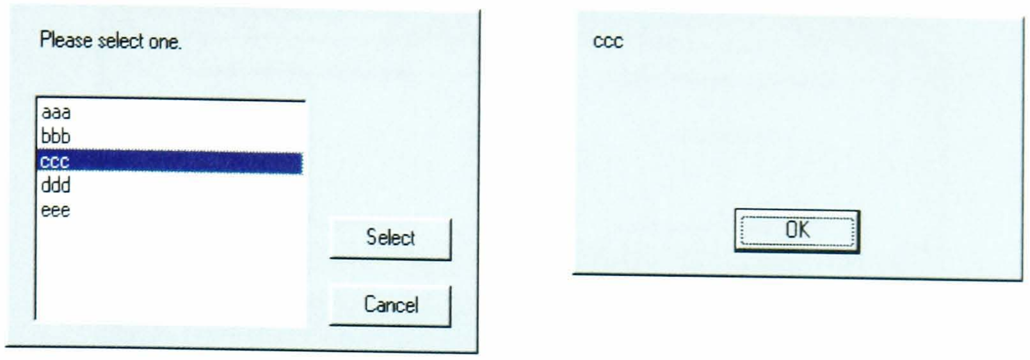


Figure 5

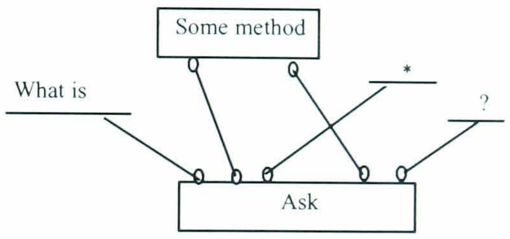
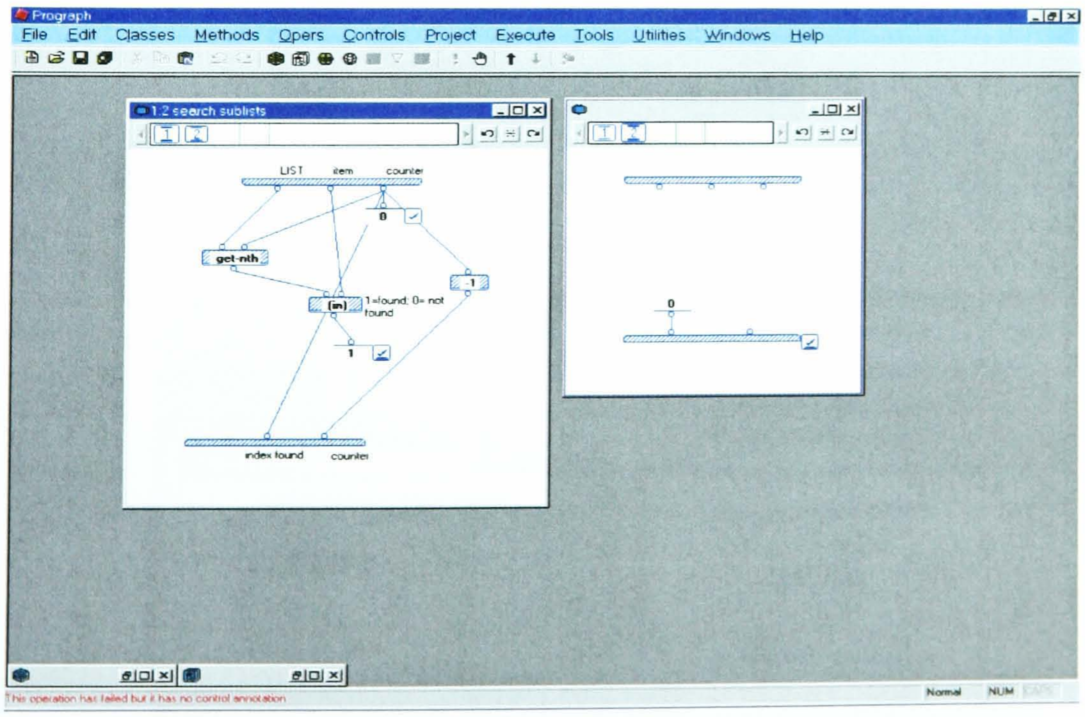


Figure 6



This operation has failed but it has no control annotation

Figure 7

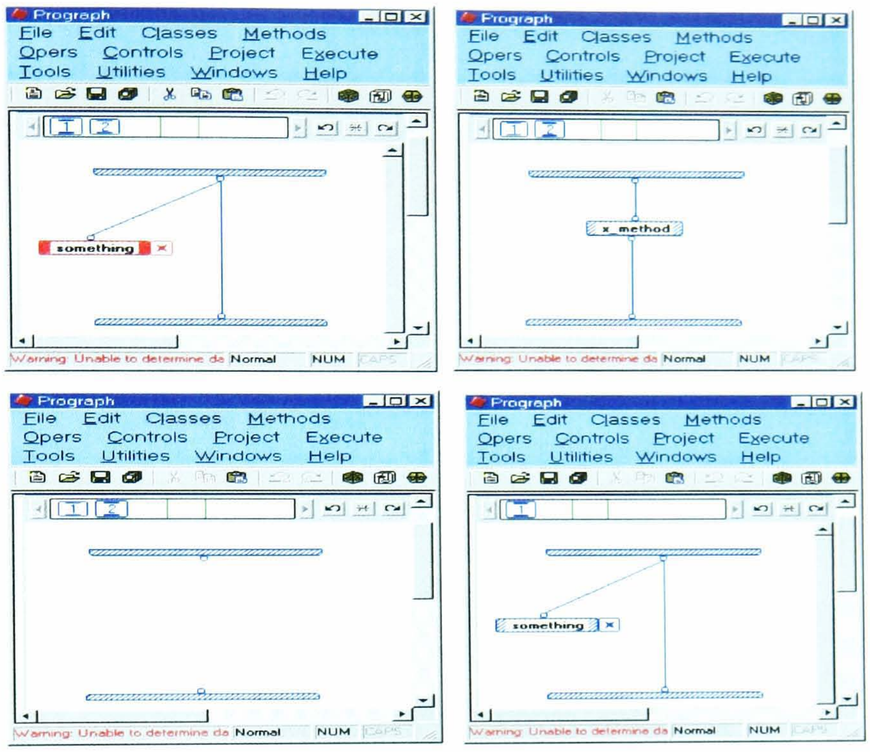

Prograph
File Edit Classes Methods
Opers Controls Project Execute
Tools Utilities Windows Help
something
x_method
Warning: Unable to determine da Normal NUM CAPS

Figure 8

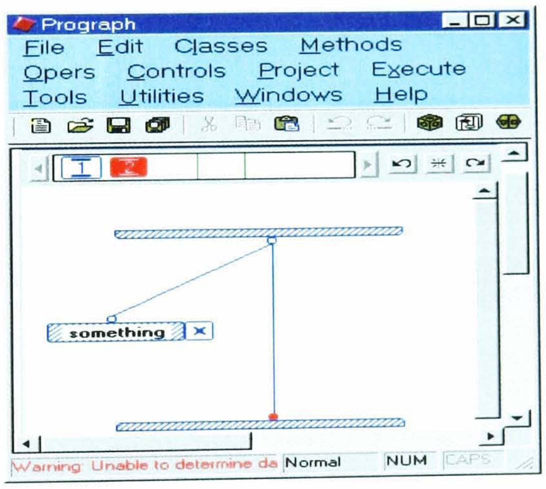

Prograph
File Edit Classes Methods
Opers Controls Project Execute
Tools Utilities Windows Help
something
Warning: Unable to determine da Normal NUM CAPS

Figure 9

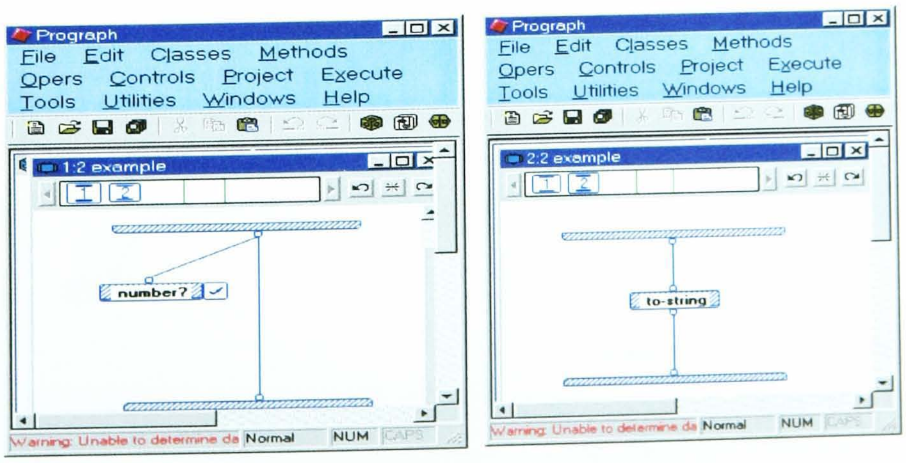

Prograph
File Edit Classes Methods
Opers Controls Project Execute
Tools Utilities Windows Help
1:2 example
number?
2:2 example
to-string
Warning: Unable to determine da Normal NUM CAPS

Figure 10

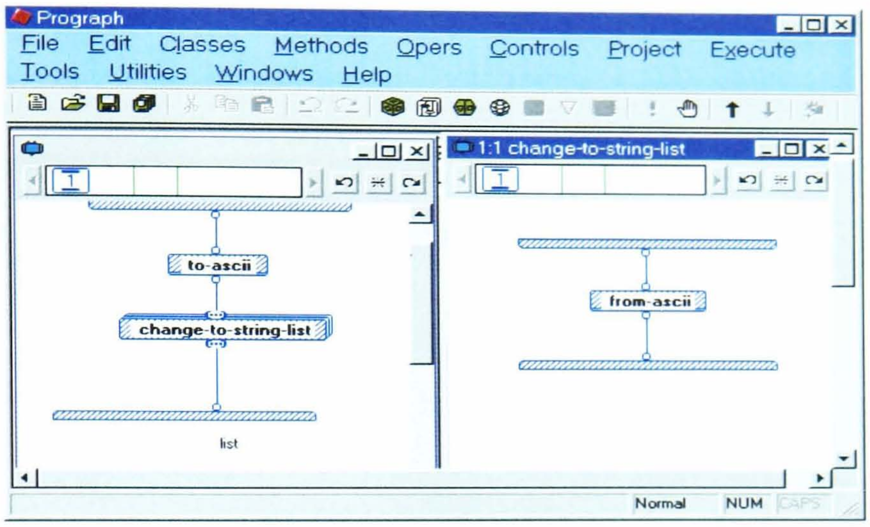
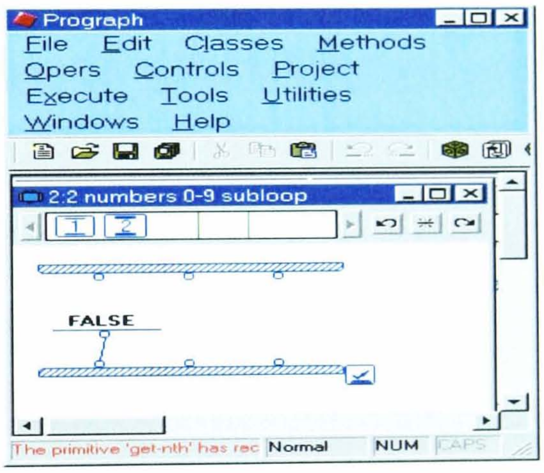
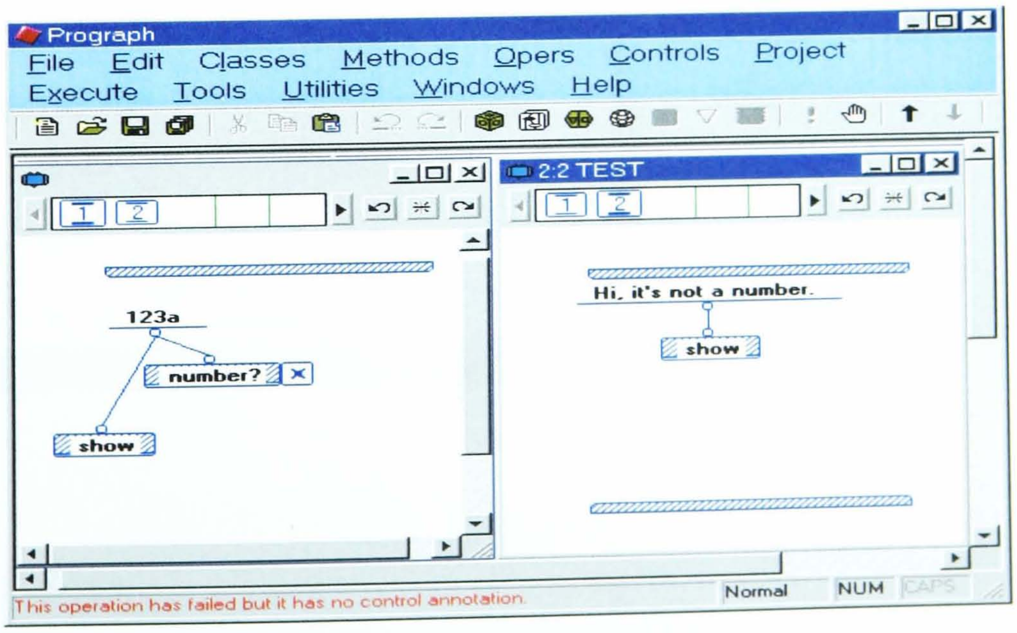

Prograph
File Edit Classes Methods Opers Controls Project Execute
Tools Utilities Windows Help
1:1 change-to-string-list
to-ascii
change-to-string-list
list
from-ascii
Normal NUM CAPS

Figure 11



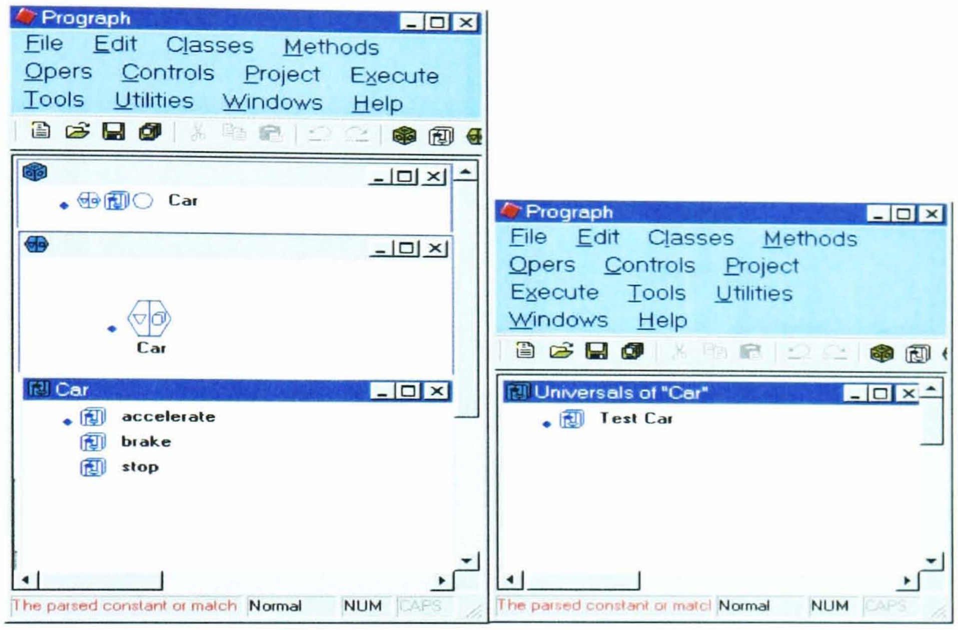
The primitive 'get-nth' has rec

Figure 12


Prograph
File Edit Classes Methods Opers Controls Project
Execute Tools Utilities Windows Help
2:2 TEST
123a
number?
show
Hi, it's not a number.
show
Normal NUM CAPS

This operation has failed but it has no control annotation.

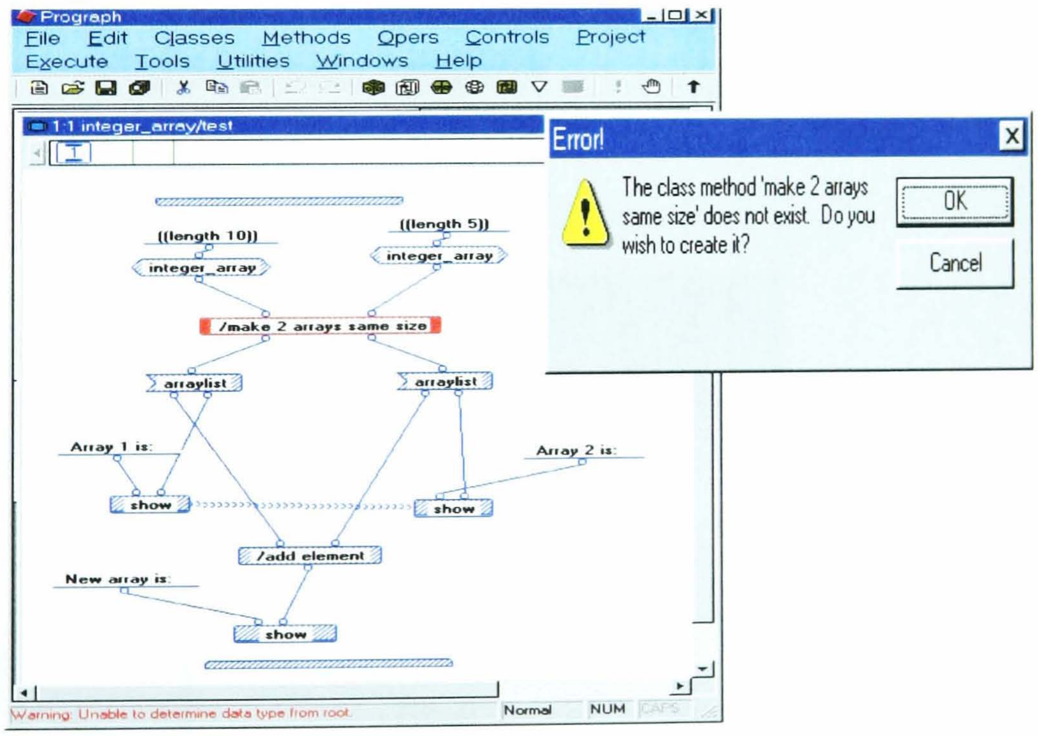
Figure 13



The parsed constant or match

The parsed constant or matcl

Figure 14



The class method 'make 2 arrays same size' does not exist. Do you wish to create it?

Warning: Unable to determine data type from root.

Figure 15

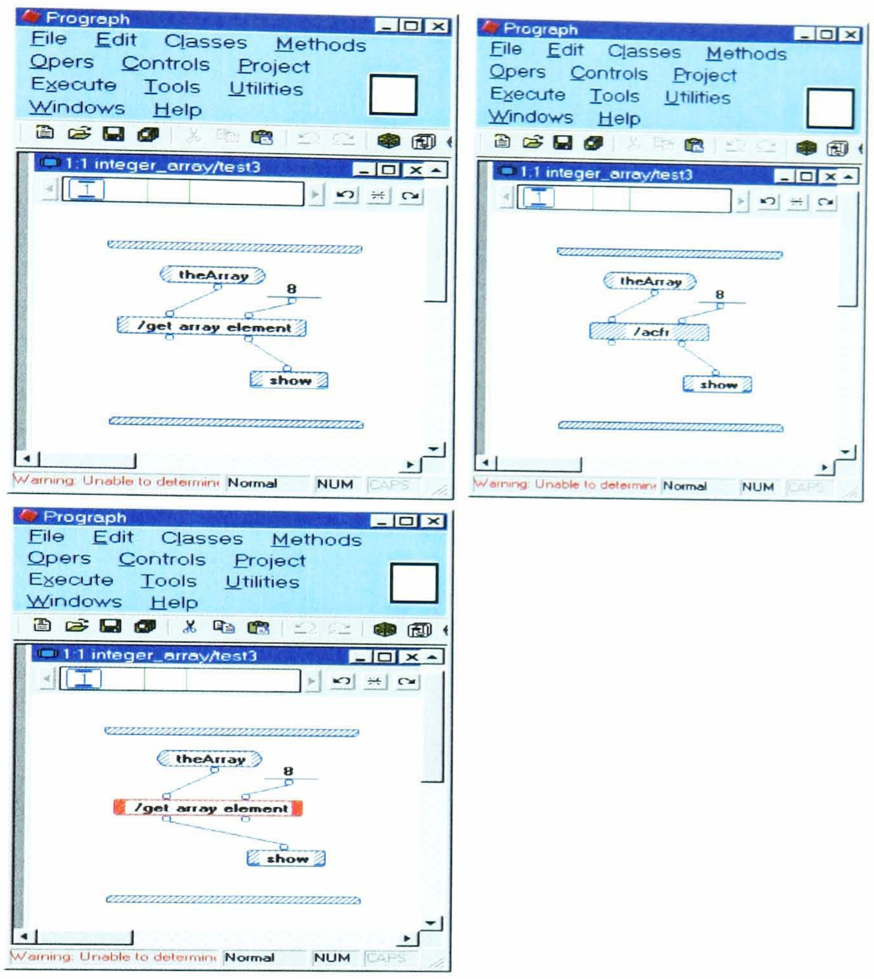


Figure 16

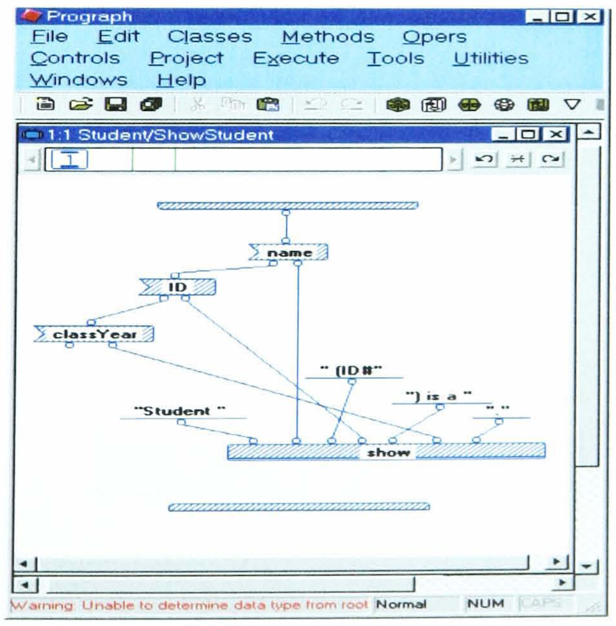
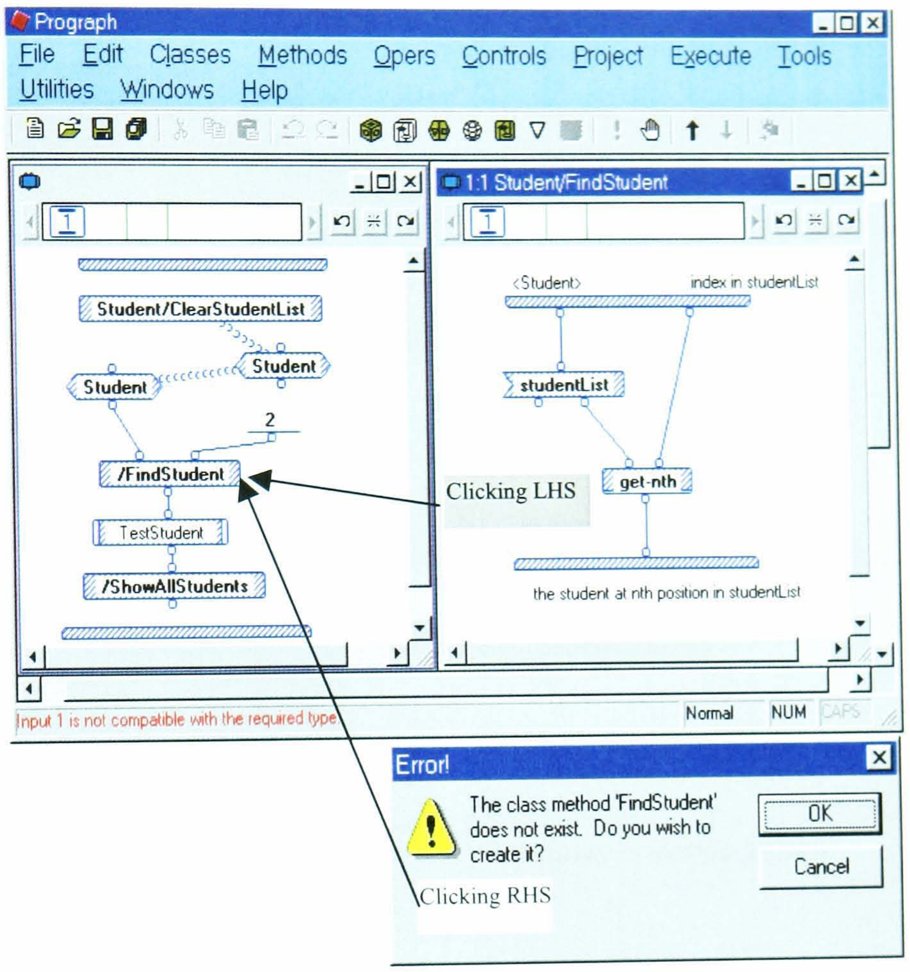

Prograph
File Edit Classes Methods Opers
Controls Project Execute Tools Utilities
Windows Help
1:1 Student/ShowStudent
name
ID
classYear
" (ID#"
") is a "
" "
"Student "
show
Warning: Unable to determine data type from root
Normal
NUM
CAPS

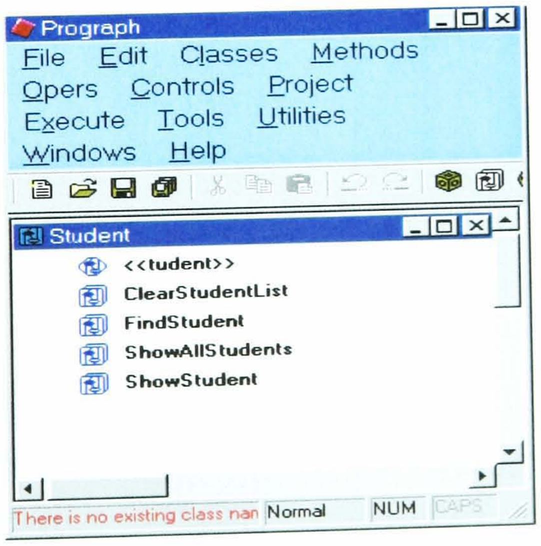
Figure 17



Input 1 is not compatible with the required type

The class method 'FindStudent' does not exist. Do you wish to create it?

Figure 18



There is no existing class nan

Figure 19

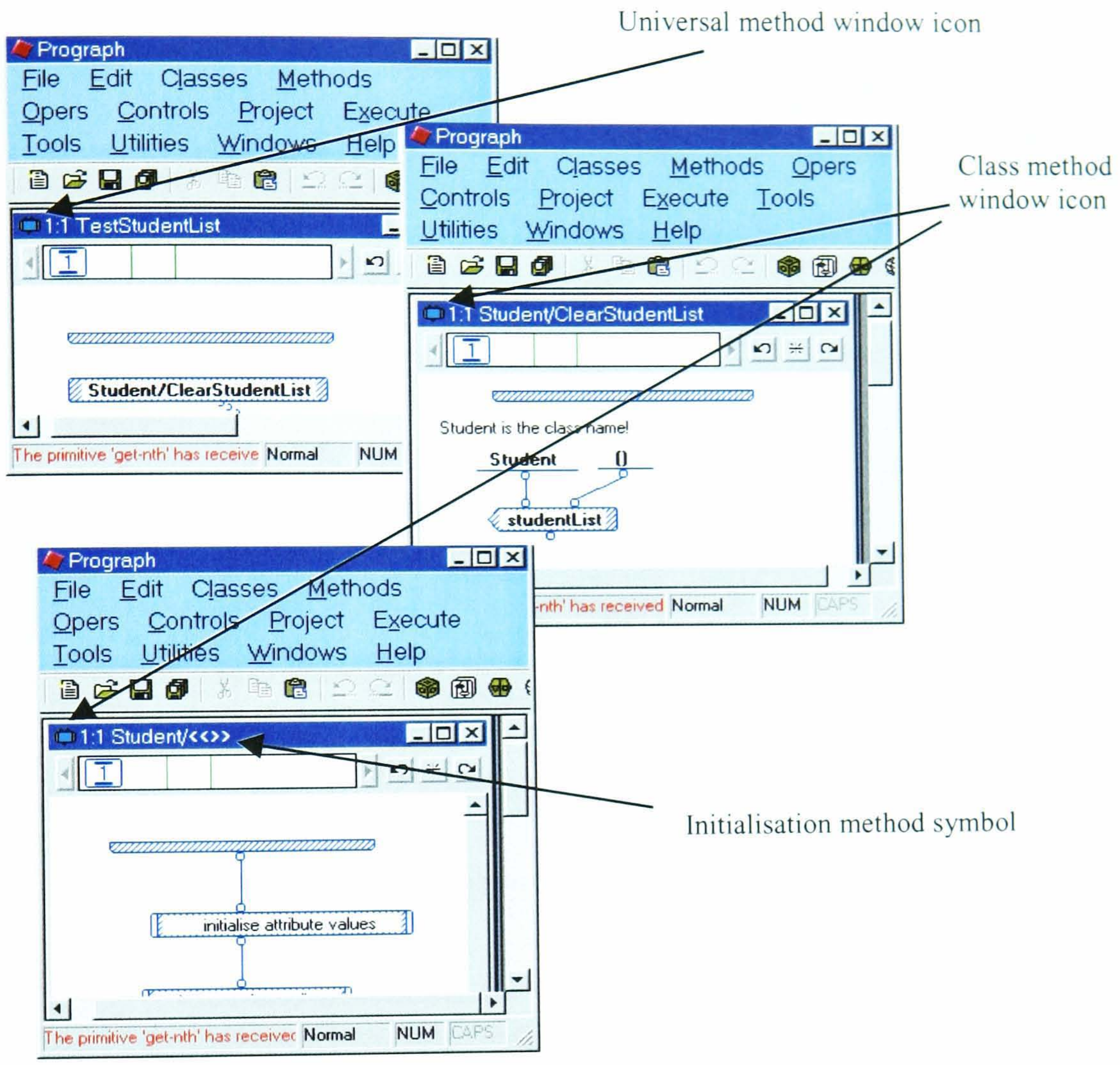
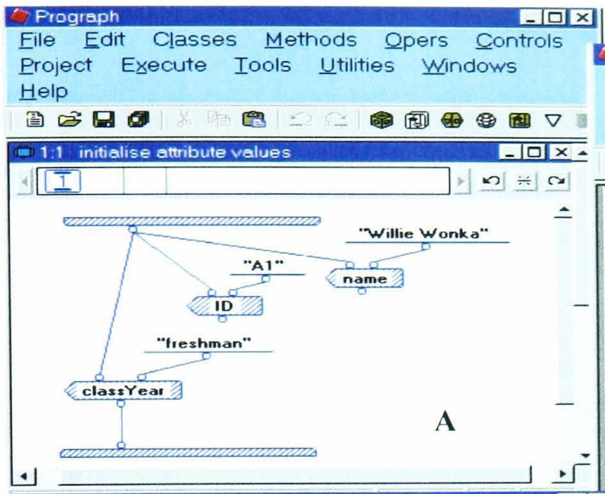
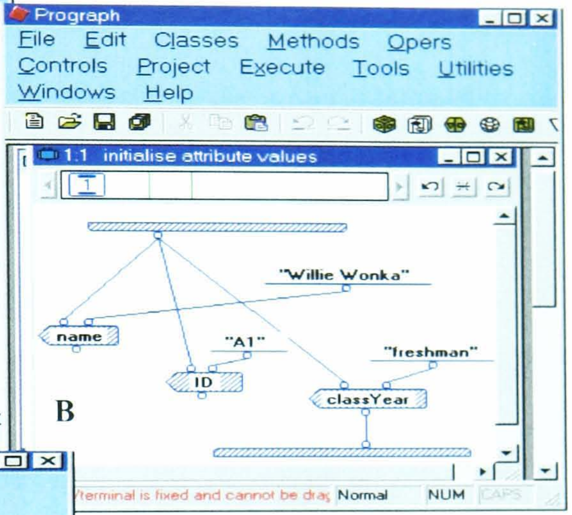
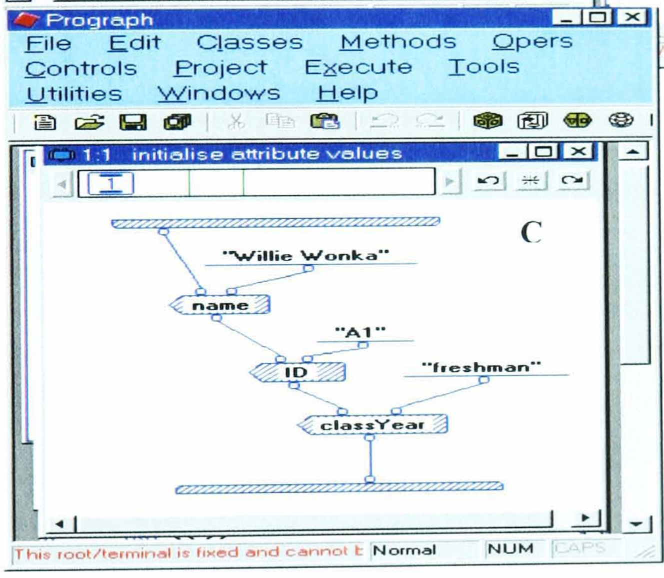

Universal method window icon
Class method window icon
Initialisation method symbol
Prograph
File Edit Classes Methods Opers Controls Project Execute Tools Utilities Windows Help
1:1 TestStudentList
Student/ClearStudentList
1:1 Student/ClearStudentList
Student is the class name!
Student
()
studentList
1:1 Student/<<>>
initialise attribute values
The primitive 'get-nth' has receive
Normal
NUM
CAPS

Figure 20


Prograph
File Edit Classes Methods Opers Controls
Project Execute Tools Utilities Windows
Help
1:1 initialise attribute values
"Willie Wonka"
"A1"
name
ID
"freshman"
classYear
A


Prograph
File Edit Classes Methods Opers
Controls Project Execute Tools Utilities
Windows Help
1:1 initialise attribute values
"Willie Wonka"
name
"A1"
"freshman"
ID
classYear
B
Normal
NUM
CAPS

terminal is fixed and cannot be drag


Prograph
File Edit Classes Methods Opers
Controls Project Execute Tools
Utilities Windows Help
1:1 initialise attribute values
C
"Willie Wonka"
name
"A1"
ID
"freshman"
classYear
Normal
NUM
CAPS

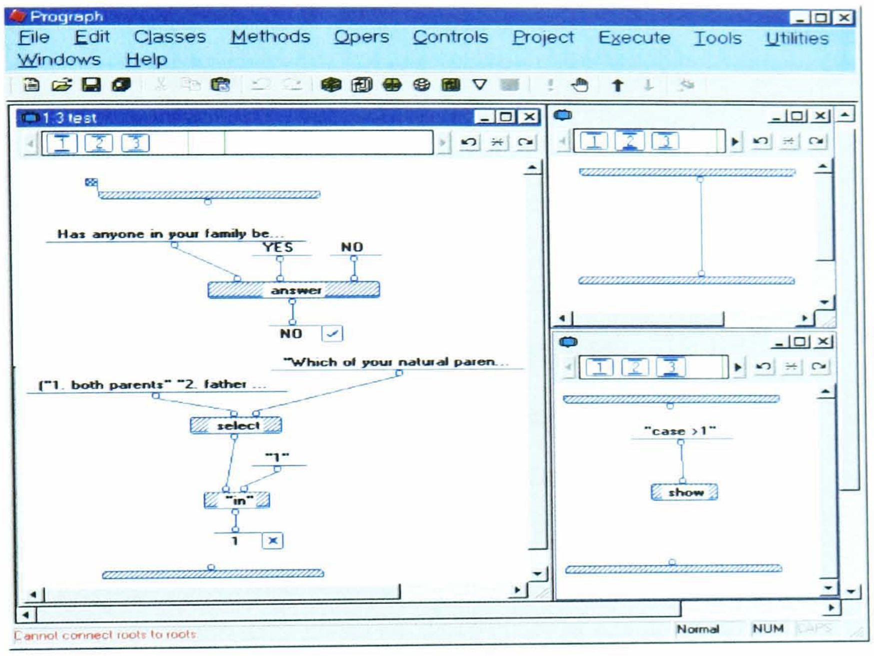
This root/terminal is fixed and cannot b

Observe these three layouts. Keeping the Prograph terminal arrangement (left for instance, right for value) and be 'neat', one has to use either A or C. Layout A and B impose no ordering, which is the virtue of data flow programming. Layout A is neat but maybe unnatural for right-handed people, who may be more comfortable with working from left to right. Layout C is probably easier for right-handed people B but it imposes ordering.

To swap or not to swap the two terminals:
C is not affected.will improve.
A will be messier.

The choice depends on whether people tend to work from left to right, even among those familiar with the left-to right writing system or those left-handed.

Figure 21


Prograph
File Edit Classes Methods Opers Controls Project Execute Tools Utilities
Windows Help
1 3 test
Has anyone in your family be...
YES
NO
answer
NO
"Which of your natural paren...
("1. both parents" "2. father ...
select
"1"
"in"
1
"case >1"
show
Normal
NUM
CAPS

Cannot connect roots to roots

APPENDIX D

Checklist and Principles for VPLs

(Chapter 6)

D-1: Six Principles for Good Diagrammatic Notations

The six principles listed in the table below are derived from findings in the literature, of which detail explanation can be found in Chapter 2 of this thesis.

Key	Description
P1	Provide appropriate means and level of abstraction.
P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
P3	Use secondary notation as appropriate.
P4	Support modification through simplicity, clarity, and flexibility.
P5	Support evaluation.
P6	Offload cognitive efforts required where possible.

D-2: Design Elements in VLM for Visual Programs

This table assigns the keys to the design elements in the VLM for visual programs in , Chapter 2 -Table 2.6, for easy referencing during the synthesis in Chapter 6.

Key	Design element	Mode	Level
M1	Font properties (font type; font size; case).	Text	Intra
M2	Names/labels.		
M3	Comments, error messages, and dialogues (call-outs).		
M4	Picture/icon size.	Spatial	
M5	Viewing angle.		
M6	Orientation of plot frame (horizontal/vertical) – as in LabVIEW control panel.		
M7	Perspective.		
M8	Size of plot frame (*x*-and *y*-axes).		
M9	Punctuation marks.	Graphics	
M10	Symbols ($, £, etc).		
M11	Treatment.		
M12	Shading.		
M13	Details of pictures/icons.		
M14	Use of colour (colour coding).		
M15	Shape of icons/objects.		
M16	Tool tip.		
M17	Numbers/letters that signal order or sequence e.g. in trees to indicate traversing path.	Text	Inter
M18	Scrollable length of the window/view.	Spatial	
M19	Layout (visibility aspect – leading, space between lines (entries) and objects).		
M20	Layout (structural aspect - traversal direction in reading diagrams).		
M21	Highlight.	Graphics	
M22	Linework in tables, organisation charts, decision trees.		
M23	Lineweight and linetype (thickness, broken, solid).		
M24	Shading.		
M25	Use of colour.		
M26	Bullets and other listing device.		
M27	Scroll bar.		
M28	Framing device (frames, boxes, lines).		
M29	Text in navigational bars.	Text	Supra
M30	Numbers/letters that signal branches of control constructs, e.g. yes/no arm, case.		
M31	Text in call-graphs and data structure trees.		
M32	Shape and orientation of windows/views unique to particular functions.	Spatial	
M33	(Consistent) position of objects across windows/views.		
M34	Background colour or texture of pictures/ icons.	Graphics	
M35	Boxes and lines around pictures or objects for reference to other parts of the program.		
M36	Pictures or icons spread over the whole document for cohesion (i.e., icons, symbols on top bar of sub-window for reference to other parts of the program.		
M37	Animation in training and debugging.		
M38	Linework in call-graphs and data structure trees.		

D-3: First-Pass Checklist Generation

This table generates 27 checkpoints from the materials in Appendix D-1 and D-2 as described in Chapter 6 –Section 6.2.1. An identification number is assigned to each checkpoint number to reflect which mode it belongs. An ID number starting with a T, Sp, and G refers to Text mode, Spatial mode, and Graphic Mode, respectively. These ID numbers are used for referencing during the process of generating the second-pass checklist in Appendix D-4.

First-pass Checklist

Check -point	ID	Brief description	Source[1]
1	T1	Use appropriate font size.	P2, M1
2	T2	Stay simple with fonts – do not use fancy and different font types.	P4, M1
3	T3	Use lower case or sentence case.	P2, M1, M2, M3, M29
4	T4	Use trigger words, meaningful names or symbols.	P2, M2
5	T5	Use easy language for dialogues, help and error messages.	P2, M3
6	T6	May use colour-coding in labels, names of different categories, types, or groups.	P3, M2, M14
7	T7	If colour-coding is used, use the colours that stand out.	P2, M14
8	T8	Do not use too many colours in the colour-coding scheme for textual messages, titles and names.	P6, M14
9	T9	Use numbers or letters as points of reference across windows/views.	P2, P4, M17
10	T10	Use standard symbols and operators (for example, using γ or <> for 'not equal').	P2, M2
11	SP1	Make sure that object size is not too small to be noticeable on a messy screen.	P2, M4
12	SP2	Avoid messy windows. Neat layouts should be achieved easily and quickly.	P2, M19
13	SP3	Avoid complex traversing rules.	P6, M20
14	SP4	Keep the position of the same object consistent in different windows as far as possible.	P2, M33
15	SP5	Avoid scrolling or keep it to minimum.	P6, M18

1. See Appendices D-1 and D-2 for keys.

First-pass Checklist (cont'd)

Check -point	ID	Brief description	Source[1]
16	G1	Use familiar icons and those that match real world objects, e.g. ✓ for ticks, ✗ or x for crosses.	P2, M13
17	G2	Use familiar or standard representations for programming constructs (such as a diamond shape for decision point as in flowcharts).	P2, M16, M15, M27, M10
18	G3	Implement coding-by-shape.	P3, M15
19	G4	Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).	P2, M-all
20	G5	Avoid too much abstraction (Do you see too many windows opened or just a few objects per window?).	P1, M32
21	G6	Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Does scrolling in a particular window/view seem endless?).	P1, M19, M18
22	G7	Use colour-coding and shading as a second means to convey a meaning. Use them in a consistent manner.	P3, M14, M25
23	G8	Make icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.	P2, M12, M13, M14, M21, M23, M24, M28
24	G9	Provide animation where appropriate such as in debugging tools.	P5, M37
25	G10	Use two-dimensional as far as possible. If three-dimensional ones must be used, use them effectively.	P2, M5, M7
26	G11	Provide some kind of tree-structure to make referencing visible.	P4, M22
27	GEN-1	Provide low fidelity tool (for example, provide functionality that supports quick and easy modification of the program code but yet does not require precision).	P4

1. See Appendices D-1 and D-2 for keys.

D-4: Second-Pass Checklist Generation

This table generates 58 checkpoints from the first pass checklist in Appendix D-3 as described in Chapter 6 –Section 6.2.2. It uses the ID numbers in Appendix D-3 and, where new ID is assigned, the same naming convention is applied. Briefly, each checkpoint in Appendix D-3 is checked against the problem tokens in Appendix C-2. Problem token numbers in Appendix C-2 that match the checkpoint being considered is recorded in the last column in the table. For problems that do not match any of the first-pass checkpoints, a new checkpoint is generated and an ID is assigned. Checkpoints 18 and 24 are excluded for the reason explained in Section 6.2.2, yielding 56 checkpoints for the second-pass checklist in total.

Second-pass checklist

Check -point	ID	Brief description	In the first pass?[1]	Reference to empirical data obtained [2]
1	T1	Use comfortable font size.	Yes	54, 88
2	T2	Stay simple with fonts – do not use fancy and different font types.	Yes	✓
3	T3	Use lower case or sentence case.	Yes	✓
4	T4	Use trigger words, meaningful names or symbols.	Yes	13, 17, 45, 71
5	T5	Use easy language for dialogues, help and error messages.	Yes	43, 61
6	T6	May use colour-coding in labels, names of different categories, types, or groups.	Yes	✓
7	T7	If colour-coding is used, use the colours that stand out.	Yes	54
8	T8	Do not use too many colours in the colour-coding scheme for textual messages, titles and names.	Yes	✓
9	T9	Use numbers or letters as points of reference across windows/views.	Yes	✓
10	T10	Use standard symbols and operators (for example, using γ or <> for 'not equal').	yes	30
11	T11	Use consistent naming convention (upper/lower case, brackets, quotation marks, etc).	no	12, 14, 22, 35, 84
12	T12	Allow users to edit a default name.	no	74

1. In Appendix D-3.
2. Refers to results from all unit studies in this thesis. The numeric code refers to problem tokens listed in Appendix C-2. The code in red refers to plus points reported. ✓means the ticked item has been implemented by Prograph VPL while ✗means the opposite.

Second-pass checklist (cont'd)

Check -point	ID	Brief description	In the first pass?[1]	Reference to empirical data obtained [2]
13	T13	Ensure that multiple floating windows/views of code are distinguishable from one another by visible and noticeable differences in titles.	no	1, 67
14	T14	Use a comma to separate items in a horizontal list rather than a space.	no	11
15	SP1	Make sure that object size is not too small to be noticeable so users do not have to search for it.	yes	✓
16	SP2	Avoid messy windows. Neat layouts should be achieved easily and quickly.	yes	1, 24, 80, 81, 83, 88, 89; Flow Study 2
17	SP3	Avoid complex traversing rules.	yes	✓; Flow Study 2
18	SP4	Keep the position of the same object consistent in different windows as far as possible.	yes	✗ (This row is to be removed.)
19	SP5	Avoid scrolling or keep it to minimum.	yes	✓; Paradigm Study and all Flow studies
20	SP6	Allow adequate separation between different parts of a graphical primitive.	no	37
21	SP7	The most current window/view must not cover the one leading to it. They are better side-by-side.	no	6, 80
22	G1	Use familiar icons and those that match real world objects, e.g. ✓ for ticks, ✗ or x for crosses.	yes	✓
23	G2	Use familiar or standard representations for programming constructs and functions.	yes	31, 33, 36, 47, 70, 75, 85 Pos-7 ✓
24	G3	Implement coding-by-shape.	yes	✗ (This row is to be removed.)
25	G4	Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).	yes	23, 25, 57, 85

1. In Appendix D-3.
2. Refers to results from all unit studies in this thesis. The numeric code refers to problem tokens listed in Appendix C-2. The code in red refers to plus points reported. ✓means the ticked item has been implemented by Prograph VPL while ✗means the opposite.

Second-pass checklist (cont'd)

Check -point	ID	Brief description	In the first pass?[1]	Reference to empirical data obtained [2]
26	G5	Avoid too much abstraction (Do you see: a) too many windows opened or b) just a few objects per window?).	yes	a) 1, 80 b) 46, 58
27	G6	Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Does scrolling in a particular window/view seem endless?).	yes	✓
28	G7	Use colour-coding and shading as a second means to convey a meaning. Use them in a consistent manner.	yes	8, 66
29	G8	Make the icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.	yes	8, 66, 69, 89 Pos-11 ✓
30	G9	Provide animation where appropriate such as in debugging tools.	yes	✓
31	G10	Use two-dimensional representations as much as possible. If three-dimensional representations must be used, use them effectively.	yes	69
32	G11	Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program.	yes	53, 80 Pos-1 ✓
33	G12	Make windows/views distinguishable from one another by making use of visible and noticeably different icons.	no	67, 88
34	G13	Avoid misleading uses by having a part in the object that looks meaningful but is meaningless or never used.	no	2, 3, 49
35	G14	Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity.	no	7, 9, 10, 38, 60, 68, 71, 84, 89
36	G15	Make appropriate use of left and right mouse-clicks for different tasks or functions on the same object (as would be expected by users). Otherwise, it only causes confusions.	no	54, 77, 86

1. In Appendix D-3.
2. Refers to results from all unit studies in this thesis. The numeric code refers to problem tokens listed in Appendix C-2. The code in red refers to plus points reported. ✓means the ticked item has been implemented by Prograph VPL while ✗means the opposite.

Second-pass checklist (cont'd)

Check -point	ID	Brief description	In the first pass?[1]	Reference to empirical data obtained [2]
37	G16	Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order of doing things (in creating objects and links, defining attributes, etc.)	no	28, 65, 88, 92, 95
38	G17	Users can add comments at any time and anywhere and are free to hide or show the comments made.	no	Pos-7, Pos-8 ✓
39	G18	Provide an icon for quickly starting a new task such as a new project. That is, make the first initial step easy to figure out.	no	5 Shortcut icons
40	G19	Provide icons for some frequently used functions for easy access (undo, execute).	no	16, 19, 39 Shortcut icons Pos-8 ✓
41	G20	Provide an undo function for all operations in manipulating objects (delete, copy, grouping).	no	18 Shortcut icons
42	G21	Automatically adjust the object to an appropriate size.	no	29 Intelligent features
43	G22	Assign only one primitive to include a few operations/tasks that are frequently used together to do a task.	no	45, 83, 88
44	G23	Allow code to be created on the fly – any time; even while the program is running.	no	Pos-3, Pos-5 ✓
45	G24	Make manipulation of objects (e.g. resizing) intuitive in both directions for paired operations, e.g., copy/delete, shrink/enlarge.	no	37
46	GEN-1	Provide a low fidelity tool.	yes	Pos-5 ✓
47	GEN-2	Remember that too much automation is not good sometimes.	no	79, 82
48	GEN-3	Do not provide a feature or function that is not meant to be available.	no	87
49	GEN-4	Make all available features work.	no	39, 43, 50, 78
50	GEN-5	Debug the application thoroughly.	no	55, 76, 95

1. In Appendix D-3.
2. Refers to results from all unit studies in this thesis. The numeric code refers to problem tokens listed in Appendix C-2. The code in red refers to plus points reported. ✓means the ticked item has been implemented by Prograph VPL while ✗means the opposite.

Second-pass checklist (cont'd)

Check -point	ID	Brief description	In the first pass?[1]	Reference to empirical data obtained [2]
51	GEN-6	Avoid hard concepts that require thinking ahead:		
		Implement an easy way to pass controls. Do no t be too restrictive.	no	15, 32, 40, 41, 51, 56, 63, 73, 75, 94
		Make loop termination simple and require no thinking ahead.	no	21, 38, 41, 42, 48, 51, 52, 60
		Ticks and crosses are difficult to use.		26, 51, 60, 62
		Make Iteration easy.	no	27, 34, 62
		Make method referencing easy.	no	81
52	GEN-7	Using graphics in the HELP document – make it visual.	no	44
53	GEN-8	Ensure Help provides a full coverage of all operations and functions.	no	64, 93
54	GEN-9	Provide a list of the exact names of operators or fuzzy search facility.	no	72
55	GEN-10	Ensure Help does not provide incorrect information.	no	90, 91
56	GEN-11	Provide adequate information in error messages.	no	78
57	GEN-12	A previous error message should either disappear or make it known that it is not applicable now.	no	20
58	GEN-13	Provide a facility to tidy up and straighten links.	no	Pos-12 ✓

1. In Appendix D-3.
2. Refers to results from all unit studies in this thesis. The numeric code refers to problem tokens listed in Appendix C-2. The code in red refers to plus points reported. ✓means the ticked item has been implemented by Prograph VPL while ✗means the opposite.

D-5: Matching Checkpoints and Design Principles.

This table list the each checkpoint in the second-pass checklist (Appendix D-4) and the six design principle(s), P1 to P6 in Appendix D-1 in which it can be categorised. Additional principles are created during this matching process when no match is found. This iterative procedure to generate the first-pass principles is described in Section 6.3.1. There are eight additional principles generated in this process, resulting in 13 principles in total.

Matching checkpoints and design principles

ID	Brief description	Design principle	Description
T1	Use comfortable font size.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T2	Stay simple with fonts – do not use fancy and different font types.	P4	Support modification through simplicity, clarity, and flexibility
T3	Use lower case or sentence case.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T4	Use trigger words, meaningful names or symbols.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T5	Use easy language for dialogues, help and error messages.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T6	May use colour-coding in labels, names of different categories, types, or groups.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
T7	If colour-coding is used, use the colours that stand out.	P3	Use secondary notation as appropriate.
T8	Do not use too many colours in the colour-coding scheme for textual messages, titles and names.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T9	Use numbers or letters as points of reference across windows/views.	P6	Offload cognitive efforts required where possible

Matching checkpoints and design principles (cont'd)

ID	Brief description	Design principle	Description
T10	Use standard symbols and operators (for example, using γ or <> for 'not equal').	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T11	Use consistent naming convention (upper/lower case, brackets, quotation marks, etc).	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T12	Allow users to edit a default name.	P4	Support modification through simplicity, clarity, and flexibility.
T13	Ensure that multiple floating windows/views of code are distinguishable from one another by visible and noticeable differences in titles.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
T14	Use a comma to separate items in a horizontal list rather than a space.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
SP1	Make sure that object size is not too small to be noticeable so users do not have to search for it.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
SP2	Avoid messy windows. Neat layouts should be achieved easily and quickly.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
SP3	Avoid complex traversing rules.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
SP4	Keep the position of the same object consistent in different windows as far as possible.	P6	Offload cognitive efforts required where possible
SP5	Avoid scrolling or keep it to minimum.	P6	Offload cognitive efforts required where possible
SP6	Allow adequate separation between different parts of a graphical primitive.	P6	Offload cognitive efforts required where possible
SP7	The most current window/view must not cover the one leading to it. They are better side-by-side.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.

Matching checkpoints and design principles (cont'd)

ID	Brief description	Design principle	Description
G1	Use familiar icons and those that match real world objects, e.g. ✓ for ticks, × or x for crosses.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G2	Use familiar or standard representations for programming constructs and functions.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G3	Implement coding-by-shape.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G4	Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G5	Avoid too much abstraction (Do you see: a) too many windows opened or b) just a few objects per window?).	P1	Provide appropriate means and level of abstraction.
G6	Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Does scrolling in a particular window/view seem endless?).	P1	Provide appropriate means and level of abstraction.
G7	Use colour-coding and shading as a second means to convey a meaning. Use them in a consistent manner.	P3	Use secondary notation as appropriate.
G8	Make the icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G9	Provide animation where appropriate such as in debugging tools.	P5	Support evaluation
G10	Use two-dimensional representations as much as possible. If three-dimensional representations must be used, use them effectively.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G11	Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.

Matching checkpoints and design principles (cont'd)

ID	Brief description	Design principle	Description
G12	Make windows/views distinguishable from one another by making use of visible and noticeably different icons.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G13	Avoid misleading uses by having a part in the object that looks meaningful but is meaningless or never used.	P11	Beware of misleading appearance.
G14	Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
G15	Make appropriate use of left and right mouse-clicks for different tasks or functions on the same object (as would be expected by users). Otherwise, it only causes confusions.	P8	Operation on devices should meet users' expectation.
G16	Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order of doing things (in creating objects and links, defining attributes, etc.)	P9	Encourage user's control and freedom.
G17	Users can add comments at any time and anywhere and are free to hide or show the comments made.	P9	Encourage user's control and freedom.
G18	Provide an icon for quickly starting a new task such as a new project. That is, make the first initial step easy to figure out.	P7	Support minimalism and economy of interactions
G19	Provide icons for some frequently used functions for easy access (undo, execute).	P7	Support minimalism and economy of interactions
G20	Provide an undo function for all operations in manipulating objects (delete, copy, grouping).	P7	Support minimalism and economy of interactions
G21	Automatically adjust the object to an appropriate size.	P7	Support minimalism and economy of interactions
G22	Assign only one primitive to include a few operations/tasks that are frequently used together to do a task.	P7	Support minimalism and economy of interactions
G23	Allow code to be created on the fly – any time; even while the program is running.	P7	Support minimalism and economy of interactions
G24	Make manipulation of objects (e.g. resizing) intuitive in both directions for paired operations, e.g., copy/delete, shrink/enlarge.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.

Matching checkpoints and design principles (cont'd)

ID	Brief description	Design principle	Description
GEN-1	Provide a low fidelity tool.	P4	Support modification through simplicity, clarity, and flexibility
GEN-2	Remember that too much automation is not good sometimes.	P7	Support minimalism.
GEN-3	Do not provide a feature or function that is not meant to be available.	P13	Consistency in provisions (e.g. of functions) and their implementation (Ensure a bug-free VPL)
GEN-4	Make all available features work.	P13	Consistency in provisions (e.g. of functions) and their implementation (Ensure a bug-free VPL)
GEN-5	Debug the application thoroughly.	P13	Consistency in provisions (e.g. of functions) and their implementation (Ensure a bug-free VPL)
GEN-6	Avoid hard concepts that require thinking ahead: - Implement an easy way to pass controls. - Do no t be too restrictive. - Make loop termination simple and require no thinking ahead - Ticks and crosses are difficult to use - Make Iteration easy - Make method referencing easy	P10	Test new features and hard concepts with real users.
GEN-7	Using graphics in the HELP document – make it visual.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
GEN-8	Ensure Help provides a full coverage of all operations and functions.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
GEN-9	Provide a list of the exact names of operators or fuzzy search facility.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
GEN-10	Ensure Help does not provide incorrect information.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.

Matching checkpoints and design principles (cont'd)

ID	Brief description	Design principle	Description
GEN-11	Provide adequate information in error messages.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
GEN-12	A previous error message should either disappear or make it known that it is not applicable now.	P12	Help content, error messages, and dialogues must be comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document.
GEN-13	Provide a facility to tidy up and straighten links.	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.

D-6: First-Pass Design Principles

This table summarises 13 principles generated in the first-pass.

Source	Key	Description
The original six design principles derived through literature review	P1	Provide appropriate means and level of abstraction
	P2	Use clearly distinguishable, familiar, and revealing representations and meaningful names in a consistent manner.
	P3	Use secondary notation as appropriate.
	P4	Support modification through simplicity, clarity, and flexibility.
	P5	Support evaluation.
	P6	Offload cognitive efforts required where possible.
Additional design principles generated from the final checklist	P7	Support minimalism and economy of interaction.
	P8	Operation on devices should meet users' expectation.
	P9	Encourage user's control and freedom.
	P10	Avoid hard concepts.
	P11	Beware of misleading appearance.
	P12	Make Help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Also, make use of graphics in Help document to ease its comprehension.
	P13	Ensure consistency in provisions (e.g. of functions) and their implementation.

D-7: Second-Pass Design Principles

This table summarises 14 principles generated in the second-pass. The procedure is described in Section 6.3.2 of Chapter 6.

Key	Description
P1	Provide appropriate means and level of abstraction
P2	Use clearly distinguishable, familiar, and revealing representations, meaningful names, and familiar functionality in a consistent manner.
P3	Use secondary notation as appropriate.
P4	Support modification through simplicity, clarity, and flexibility.
P5	Support evaluation (by providing suitable functionality).
P6	Offload cognitive efforts required where possible.
P7	Support minimalism and economy of interactions
P8	Operation on devices should meet users' expectation.
P9	Encourage user's control and freedom.
P10	Avoid hard concepts.
P11	Make Help content, error messages, and dialogues comprehensible, relevant, sufficient, and up to date. Make use of graphics in Help document to ease its comprehension.
P12	Ensure consistency in provisions (e.g. of functions) and their implementation.
P13	Ensure consistency in the ways things are done.
P14	Prevents or corrects errors (by providing appropriate automated functionality and by avoiding misleading appearance).

D-8: Refined second-pass checklist

The second-pass checklist in Appendix D-4 has been refined in the refinement phase (see Section 6.3.2 of this thesis), in which two more checkpoints are added to it, yielding 58 checkpoints in total.

1. Use comfortable font size.
2. Stay simple with fonts – do not use fancy and different font types.
3. Use lower case or sentence case.
4. Use trigger words, meaningful names or symbols.
5. Use easy language for dialogues, help, text and error messages.
6. May use colour-coding in labels, names of different categories, types, or groups.
7. If colour-coding is used, use the colours that stand out.
8. Do not use too many colours in the colour-coding scheme for textual messages, titles and names.
9. Use numbers or letters as points of reference across windows/views.
10. Use standard symbols and operators (for example, using γ or <> for 'not equal').
11. Use consistent naming convention (upper/lower case, brackets, quotation marks, etc.).
12. Allow users to edit a default name.
13. Ensure that multiple floating windows/views of code are distinguishable from one another by visible and noticeable differences in titles.
14. Use a comma to separate items in a horizontal list rather than a space.
15. Make sure that object size is not too small to be noticeable so users do not have to search for it.
16. Avoid messy windows. Neat layouts should be achieved easily and quickly.
17. Avoid complex traversing rules.
18. Avoid scrolling or keep it to minimum.
19. Allow adequate separation between different parts of a graphical primitive.
20. The most current window/view must not cover the one leading to it. They are better side-by-side.
21. Use familiar icons and those that match real world objects, e.g. ✓ for ticks, ×or x for crosses.
22. Use familiar or standard representations for programming constructs and functions.
23. Exploit standard convention (for example, branching or small section of code is in top-down fashion rather than in bottom-up fashion).
24. Avoid too much abstraction (Do you see: a) too many windows opened or b) just a few objects per window?).
25. Avoid too little abstraction (Do you see objects dispersed everywhere in the same window which could have been grouped? Does scrolling in a particular window/view seem endless?).
26. Use colour-coding and shading as a secondary means to convey a meaning. Use them in a consistent manner.

27. Make the icons/objects look distinctive (distinctly different). Use colour, highlights, shading, lineweight, and framing to promote discriminability. Make them noticeable.
28. Provide animation where appropriate such as in debugging tools.
29. Use two-dimensional representations as much as possible. If three-dimensional representations must be used, use them effectively.
30. Provide some kind of tree-structure to make referencing visible. For example, provide a visible, 2-way class/method navigation tool such as tree-structure for method referencing or provide a list of methods created so far in the program.
31. Make windows/views distinguishable from one another by making use of visible and noticeably different icons.
32. Avoid misleading users by having a part in the object that looks meaningful but is meaningless or never used.
33. Make all parts in an object role expressive. Icons must reflect the intended meanings. Graphical primitives should have their visual identity.
34. Make appropriate use of left and right mouse-clicks for different tasks or functions on the same object (as would be expected by users). Otherwise, it only causes confusion.
35. Avoid representations that impose a certain programming style on users, e.g., order. Allow flexible order for doing things (in creating objects and links, defining attributes, etc.).
36. Users can add comments at any time and anywhere and are free to hide or show the comments made.
37. Provide an icon for quickly starting a new task such as a new project. That is, make the initial step easy to figure out.
38. Provide icons for some frequently used functions for easy access (undo, execute).
39. Provide an undo function for all operations in manipulating objects (delete, copy, grouping).
40. Automatically adjust the object to an appropriate size.
41. Assign only one primitive to include a few operations/tasks that are frequently used together to do a task.
42. Allow code to be created on the fly – any time; even while the program is running.
43. Make manipulation of objects (e.g. resizing) intuitive in both directions for paired operations, e.g., copy/delete, shrink/enlarge.
44. Provide a low fidelity tool.
45. Remember that too much automation is not good sometimes.
46. Do not provide a feature or function that is not meant to be available.
47. Make all available features work.
48. Debug the application thoroughly.
49. Avoid hard concepts that require thinking ahead in:
    - Passing controls.
    - Terminating a loop.
    - Performing iteration.
    - Referencing.
50. Use graphics in the HELP document – make it visual.
51. Ensure Help provides a full coverage of all operations and functions.

52. Provide a list of the exact names of operators or fuzzy search facility.
53. Ensure Help does not provide incorrect or outdated information.
54. Provide adequate information in error messages.
55. A previous error message should either disappear or make it known that it is not applicable now.
56. Provide a facility to tidy up and straighten links.
57. Be consistent in the way by which variable values are passed, dealing with data types, in assigning primitive names, in applying rules – make it applicable in all situations, etc.
58. Provide automatic facilities such as error-checking, garbage collector, and spell-check facility.

REFERENCES

Adler, A., Gujar, A., Harrison, B., O'Hara, K., & Sellen, A. (1998). A diary study of work-related reading: Design implications for digital reading devices. In *Proceedings of CHI'98 Conference on Human Factors in Computing Systems* (pp. 241-248). ACM Press.

Adler, P. (1984). The sociologist as celebrity: The role of the media in the field research. *Qualitative Sociology, 7*, 319-326.

Adler, P. A. & Adler, P. (1994). Observational techniques. In N. K. Denzin & Y. S. Lincoln (Eds.), *Handbook of qualitative research* (pp.377-392). Thousand Oaks: Sage.

Albizuri-Romero, M. B. (1984). A graphical abstract programming language. *ACM SIGPLAN Notices, 19*(1), 14-23.

Andre, T. S. (2001). The user action framework: A reliable foundation for usability engineering support tools, *International Journal of Human-Computer Studies, 54*, 107-136.

Anjaneyulu, K. S. R. & Anderson, J. R. (1992). The advantage of data flow diagrams for beginning programming. In C. Frasson, G. Gauthier, & G.I. McCalla (Eds.), *Intelligent Tutoring Systems: Second International Conference, ITS' 92* (pp.585-592).

Baecker, R. M., Grudin, J., Buxton, W. A. S., & Greenberg, S. (1995). *Readings in human-computer interaction: Toward the year 2000* (2nd ed.) (pp.411-423). San Francisco: Morgan Kaufmann.

Bailey, R. W. (1999). Heuristic evaluation. *UI design update newsletter - May 1999*, Retrieved August 8, 2002, from http://www.humanfactors.com/downloads/may99.asp.

Bailey, R. W. (2001). Heuristic evaluation vs user testing, *UI Design Update Newsletter— January 2001*, Retrieved December 10, 2001, from http://www. humanfactors.com/library/jan001.htm.

Bell, B., Citrin, W., Lewis, C., Rieman, J., Weaver, R., Wilde, N. & Zorn, B. (1992). *The programming walkthrough: A structured method for assessing the writability of programming languages* (Technical Report CU-CS-577-92). Department of Computer Science, University of Colorado.

Bergantz, D. & Hassell, J. (1991). Information relationships in PROLOG programs: How do programmers comprehend functionality? *International Journal of Man-Machine Studies, 35*, 313-328.

Bevan, N. (2002). *Cost effective user-centered design on ISO 13407* (Tutorial notes). The 11th Annual Conference of Usability Professionals Association, Orlando, Florida.

Bivins, T. & Ryan, W. E. (1991). How to produce creative publications: Traditional techniques & computer applications. Lincolnwood, Illinois: NTC Business Books.

Blackwell, A. (1996). Metacognitive theories of visual programming: What do we think we are doing? In *Proceedings of the IEEE Symposium on Visual Languages* (pp. 240-246). CA: IEEE Computer Society.

Blackwell, A. F. (2000). *Human computer interaction notes on advanced graphics & HCI: Part II course 2000-2001*, University of Cambridge. Retrieved January 16, 2002, from http://www.cl.cam.ac.uk/Teaching/2001/AGraphHCI/HCI/hci2001.pdf.

Blackwell, A.F. (2002). *Cognitive Dimensions of Notations resource site*. Retrieved January 16, 2002, from http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/.

Blackwell, A. F. & Green, T. R. G. (2000). A Cognitive Dimensions Questionnaire optimised for users, In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group* (pp.137-152). Cosenza, Italy: Memoria.

Blackwell, A. F., Whitley, K., Good, J. & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review, 15*(1), 95-113.

Blaiwes, A. S. (1974). Formats for representing procedural instructions. *Journal of Applied Psychology, 59*(6), 683-686.

Bonar, J. & Liffick, B. W. (1990). A visual programming language for novices. In S.K. Chang (Ed.), *Principles of Visual Programming Systems* (pp.326-366). NJ: Prentice-Hall.

Bowles, K. L. (1977). *Microcomputer problem solving using Pascal*. NY: Springer Verlag.

Breakwell, G. M. & Wood, P. (2000). Diary techniques. In G. M. Breakwell, S. Hammond, & C. Fife-Schaw (Eds.), *Research methods in psychology* (pp.294-302). London: Sage.

Britton, C. & Jones, S. (1999). The untrained eye: How languages for software specification support understanding in untrained users. *Human-Computer Interaction, 14*, 191-244.

Britton, C. & Kutar, M. (2001). Cognitive Dimensions profiles: A cautionary tale. In G. Kadoda (Ed.), *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group* (pp.265-274). Sheffield: Print Unit.

Brodie, J. & Chattratichart, J. (2002). Establishing design guidelines for a better online shopping experience. In *Proceedings of the 11th Annual Conference of the Usability Professional's Association* (pp. 51). UPA.

Brooke, J. B. & Duncan, K. D. (1980). Experimental studies of flowchart use at different stages of program debugging. *Ergonomics, 23*(11), 1057-1091.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18*(6), 543-554.

Brown, B., Sellen, A., & O'Hara, K. (2000). A diary study of information capture in working life. In *Proceedings of CHI'2000 Conference on Human Factors in Computer Systems* (pp. 438-445). ACM Press.

Bruyn, S. (1966). The human perspective in sociology: The methodology of participant observation. Englewood Cliffs, NJ: Prentice-Hall.

Burnett, M. & Baker, M. J. (1994). A classification system for visual programming languages. *Journal of Visual Languages and Computing, 5*, 287-300.

Catarci, T. & Santucci, G. (1995). Are visual query languages easier to use than traditional ones? An experimental proof. In M. A. R. Kirby, A. J. Dix & J. E. Finlay (Eds.), *People and Computer X* (pp. 323-338).

Chattratichart, J. & Brodie, J. (2002a). Establishing design guidelines for a better online shopping experience. In *Proceedings of the 11th Annual Conference of the Usability Professional's Association* (pp. 51). UPA.

Chattratichart, J. & Brodie, J. (2002b). Extending the Heuristic Evaluation method through contextualisation. In the *Proceedings of the 46th Annual Meeting of the Human Factors and Ergonomics Society* (pp. 641-645). HFES.

Chattratichart, J. & Brodie, J. (2003). HE-Plus: Toward usage-centered expert review for website design. In L. L. Constantine (Ed.), *Proceedings of forUSE 2003, Second International Conference on Usage-Centered Design* (pp. 155-169). Massachusetts: Ampersand Press.

Chattratichart, J., Cave, D. & Vaduva, A. (2003). Learning and doing 'expert evaluation': A teaching dilemma. In L. L. Constantine (Ed.), *Proceedings of forUSE 2003, Second International Conference on Usage-Centered Design* (pp. 27-35). Massachusetts: Ampersand Press.

Chattratichart, J. & Kuljis, J. (2001). Some evidence for graphical readership, paradigm preference, and the Match-Mismatch conjecture in graphical programs. In G. Kadoda (Ed.), *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group* (pp.173-189). Sheffield: Print Unit.

Chin, J. P., Herring, R.D., & Familant, M. E. (1992). A usability and diary study assessing the effectiveness of call acceptance lists. In *Proceedings of the 36th Annual Meeting of the Human Factors and Ergonomics Society* (Vol. 1, pp. 216-220). HFES.

Clarke, S. (2001). Evaluating a new language. In G. Kadoda (Ed.), *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group* (pp.275-289). Sheffield: Print Unit.

Corritore, C. L. & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies, 50*, 61-83.

Corti, L. (2002). *Using diaries in social research.* Retrieved November 24, 2002, from http://www.soc.surrey.ac.uk/sru/SRU2.htm.

Cox, K. (2000). Cognitive Dimensions of Use Cases-Feedback from a student questionnaire. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group* (pp.99-121). Cosenza: Memoria.

Cunniff, N., Taylor, R. P., & Black, J. B. (1989). Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 419-429). Lawrence Erlbaum Associates.

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

Davies, S. P. (2000). Expertise and the comprehension of object-oriented programs. In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group* (pp.61-65). Cosenza, Italy: Memoria.

Dutt, A., Johnson, H. & Johnson, P. (1994). Evaluating evaluation methods, In *Proceedings of CHI'94 Conference on Human Factors in Computing Systems* (pp. 109-121). ACM Press.

Eisenstadt, M. & Breuker, J. (1992). Naïve iteration: An account of the conceptualizations underlying buggy looping program. In M. Eisenstadt, M. T. Keane, & T. Rajan (Eds.), *Novice programming environments: Exploration in human-computer interaction and artificial intelligence* (pp.173-188). Hove: Lawrence Erlbaum Associates.

Ekstrom, R. B., French, J. W., Harman, H. H., & Dermen, D. (1976). *Manual for the kit of factor-referenced cognitive tests.* Princeton, NJ: Educational Testing Service.

Ellis, C. (1991). Sociological introspection and emotional experience. *Symbolic Interaction, 14*, 23-50.

Eysenck, M. W. & Keane, M. T. (1992). *Cognitive psychology: A student's handbook.* Hove: Lawrence Erlbaum Associates.

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

Frei, H. P., Weller, D. L. & Williams, R. (1978). A graphical-based programming-support system. *Computer Graphics (ACM SIGGRAPH), 12*(3), 43-49.

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

Glinert, E. P. (1990). Nontextual programming environments. In S. K. Chang (Ed.), *Principles of visual programming systems* (pp.162-169). NJ: Prentice-Hall.

Glinert, E. P. & Tanimoto, S. L. (1990). Pict: An interactive graphical programming environment. In E. P. Glinert (Ed.), *Visual programming environments: Paradigms and systems* (pp. 265-283). IEEE Computer Society Press.

Good, J. (1999). Programming paradigms, information types and graphical representations: Empirical investigations of novice program comprehension. Unpublished PhD Dissertation, Edinburgh University, UK.

Gould, J. D. (1995). How to design usable systems. In R. M. Baecker, J. Grudin, W. A. S. Buxton, & S. Greenberg (Eds.), *Readings in human-computer interaction: Toward the year 2000.* (2nd ed.) (pp.93-121). San Francisco: Morgan Kaufmann.

Gould, J. D. & Lewis, C. H. (1985). Designing for usability – Key principles and what the designers think. *Communications of the ACM, 28*(3), 300-311.

Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology, 50*, 93-109.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith & T. R. G. Green (Eds), *Human Interaction with Computers* (pp.271-320). London: Academic Press.

Green, T. R. G. (1982). Pictures of programs and other processes, or how to do things with lines. *Behaviour and Information Technology, 1*(1), 3-36.

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and Computers V* (pp.443-460).

Green, T. R. G. (1990). Programming languages as information structures. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of Programming* (pp.117-137). London: Academic Press.

Green, T. R. G. & Blackwell, A. F. (1998). *Cognitive Dimensions of Information Artefacts: A Tutorial.* Retrieved January 16, 2002, from http://www.ndirect.co.uk/~thomas.green/workStuff /Papers/.

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 J. Koenemann-Belliveau, T. G. Moher & S. P. Robertson (Eds.), *Empirical Studies of Programmers: Fourth Workshop* (pp.121-141). Norwood, NJ: Ablex.

Green, T. R. G., Sime, M. E. & Fitter, M. J. (1981). The art of notation. In M. J. Coombs & J. Alty (Eds.), *Computing skills and the user interface* (pp. 221-251). London: Academic Press.

Halewood, K. & Woodward, M. R. (1993). A uniform graphical view of the program construction process: GRIPSE. *International Journal of Man-Machine Studies, 38,* 805-837.

Hitchcock, D. & Taylor, A. (2003). Simulation for inclusion…true user-centred design?, *Include 2003 Conference Proceedings* (pp.105-110). London: Royal College of Art.

Hoc, J-M (1989). Do we really have conditional statements in our brains? In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp.179-190). Hillsdale, NJ: Lawrence Erlbaum Associates.

Horn, R. E. (1998). *Visual language: Global communication for the 21st century.* Bainbridge Island, WA: Macro VU Press.

Horn, R. E. (1999). Information design: Emergence of a new profession. Retrieved May 21, 2003, from http://www.stanford.edu/~rhorn/Horn-InfoDesignChapter.html.

Houde, S. & Sellman, R. (1994). In search of design principles for programming environments. In *Proceedings of CHI'94 Conference on Human Factors in Computing Systems* (pp. 424-430). ACM Press.

Jackson, M. A. (1975). *Principles of Program Design.* London: Academic Press.

Jonsson, D. (2001). *The Pancode/Boxchart home page*. Retrieved May 21, 2003, from http://www.panis.com.

Jordan, P. W. (2000). Designing pleasurable products: An introduction to the new human factors. London: Taylor & Francis.

Jorgensen, D. L. (1989). Participant observation: A methodology for human studies. Newbury Park, CA: Sage.

Kahney, H. (1992). Some pitfalls in learning about recursion. In M. Eisenstadt, M. T. Keane, & T. Rajan (Eds.), *Novice programming environments: Exploration in human-computer interaction and artificial intelligence* (pp.155-172). Hove: Lawrence Erlbaum Associates.

Kammann, R (1975). The comprehension of printed instructions and the flowchart alternative. *Human Factors*, *17*, 183-191.

Karat, C.-M. (1994). A comparison of user interface evaluation methods. In J. Nielsen & R. L. Mack (Eds.), *Usability Inspection Methods* (pp.203-233). New York: John Wiley & Sons.

Karat, C-M., Campbell, R., and Fiegel, T. (1992). Comparison of empirical testing and walkthrough methods in user interface evaluation. In *Proceedings of CHI'92 Conference on Human Factors in Computing Systems* (pp. 397-404). ACM Press.

Keppel, G. (1991). Design and analysis: *A researcher's handbook* (3rd ed.). Englewood Cliffs, NJ: Prentice Hall.

Kessler, C. M. & Anderson, J. R. (1989). Learning flow of control: Recursive and iterative procedures. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp.229-260). Hillsdale, NJ: Lawrence Erlbaum Associates.

Kessner, M., Wood, J., Dillon, R. F. & West, R. L. (2001). On the reliability of usability testing. In *Proceedings of CHI'2001 Conference on Human Factors in Computing Systems Extended Abstracts* (pp. 97-98). ACM Press.

Kirakowski, J. & Corbett, M. (1990). *Effective methodology for the study of HCI.* Mew York: Elsevier.

Klare, G. R., Nichols, W. H. & Sherford, E. H. (1957). The relationship of typographical arrangement to the learning of technical material. *Journal of Applied Psychology*, *41*, 41-45.

Kostelnick, C. & Roberts, D. D. (1998). *Designing visual language*. Boston: Allyn and Bacon.

Kotarba, J. A. (1977). The chronic pain experience. In J. D. Douglas & J. M. Johnson (Eds.), *Existential sociology* (pp.257-272). Cambridge: Cambridge University Press.

Kotarba, J. A. & Fontana, A. (1984). *The existential self in society*. Chicago: University of Chicago Press.

Krieger, S. (1985). Beyond subjectivity: The use of the self in social science. *Qualitative Sociology*, *8*, 309-324.

Kutar, M., Britton, C., & Wilson, J. (2000). Cognitive Dimensions: An experience report, In A. F. Blackwell & E. Bilotta (Eds.), *Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group* (pp.81-98). Cosenza, Italy: Memoria.

Letovsky, S. (1986). Cognitive processes in program comprehension. In E. Soloway & S. Iyengar (Eds.), *Empirical Studies of Programmers: First Workshop* (pp.58-79). Norwood, NJ: Ablex.

Lewis, C. & Olson, G. M. (1987). Can principles of cognition lower the barriers to programming? In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp.248-263). Norwood, NJ: Ablex.

Lincoln, Y. S. & Guba, E. G. (1985). *Naturalistic Inquiry*. Beverly Hills, CS: Sage.

Lindgaard, G. (1994). Usability Testing and System Evaluation: A Guide for Designing Useful Computing Systems. Chapman and Hall.

Marcus, A. (1992). Graphic design for electronic documents and user interfaces. NY: ACM Press.

Mayhew, D. J. (1999). The usability engineering lifecycle: A practitioner's handbook for user interface design. San Francisco: Morgan Kaufmann.

Miller, G.A. (1956). The magic number seven plus or minus two: Some limits of our capacity for information processing. *Psychological Review, 63*(2), 81-87.

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies, 6*, 237-260.

Modugno, F. (1996). *Cognitive Dimensions and an empirical evaluation: Lessons learned.* Retrieved January 16, 2002, from http://www.cs.cmu.edu/afs/cs.cmu.edu/user/fmm/public/www/chi96-brs.html.

Modugno, F. & Myers., B. (1994). *Pursuit: Visual programming in a visual domain* (Technical Report CMU-CS-94-109). Carnegie Mellon University. Retrieved January 16, 2002 from http//cs-tr.cs.cornell.edu:80/Dientst/UI/1.0/Display/ncstrl.cmu/CS-94-109.

Moher, T. G., Mak, D. C., Blumenthal, B., & Laventhal, L. M. (1993). Comparing the comprehensibility of textual and graphical programs: The case of Petri Nets. In C. R. Cook, J. C. Scholtz, & J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop* (pp.137-157). Norwood, NJ: Ablex.

Molich, R & Robin, J. (2003). Comparative expert reviews. In Proceedings of CHI'2003 Conference on Human Factors in Computing Systems Extended Abstracts (pp.1060-1061). ACM Press.

Molich, R., Thomsen, A. D., Karyukina, B., Schmid, L., Ede, M., van Oel, W., & Arcuri, M. (1999). Comparative evaluation of usability tests. In *Proceedings of CHI'99 Conference on Human Factors in Computing Systems Extended Abstracts* (pp. 83-84). ACM Press.

Moore, P. & Charles, P. C. (1985). *Disguised: A True Story*. Word Publishing

Myers, B. (1990). Taxonomies of visual programming and program visualisation. *Journal of Visual Languages and Computing, 1*, 97-123.

Myers, B. (n.d.). *Usability issues in programming languages*. Retrieved August 12, 2002, from http://www-2.cs.cmu.edu/~NatProg/Principles.

Nassi, I. & Shneiderman, B. (1973). Flowchart techniques for Structured Programming, *ACM Sigplan Notices*, *8*(8), 12-26.

Nielsen, J. (1993). *Usability Engineering*. London: Academic Press.

Nielsen, J. (1994). Heuristic evaluation. In J. Nielsen & R. L. Mack, Eds. *Usability Inspection Methods*, pp. 25-62. NY: John-Wiley & Sons.

Nielsen, J. & Molich, R. (1990). Heuristic evaluation of user interfaces. In *Proceedings of CHI'90 Conference on Human Factors in Computing Systems* (pp. 249-256). ACM Press.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental Models*. NJ: Lawrence Erlbaum Associates.

O'hara, K. & Perry, M. (2001). Shopping anytime anywhere. In Proceedings of CHI'2001 Conference on Human Factors in Computing Systems Extended Abstracts (pp. 345-346). ACM Press.

Omerod, T. C., Manktelow, K. I., Robson, E. H., & Steward. A. P. (1986). Content and representation effects with reasoning tasks in PROLOG form. *Behaviour and Information Technology*, *5*(2), 157-168.

Palen, L. & Salzman, M. (2002). *Voice-mail diary studies for naturalistic data capture under mobile conditions*. Retrieved November 24, 2002, from http://www.cs.colorado.edu/~palen/Papers/voice.pdf.

Pane, J. F. & Myers, B. A. (1996). *Usability Issues in the Design of Novice Programming Systems* (Technical Report CMU-CS-96-132). School of Computer Science, Carnegie Mellon University.

Pane, J. F. & Myers, B. A. (2000). Improving user performance on Boolean queries. In *Proceedings of CHI'2000 Conference on Human Factors in Computing Systems Extended Abstracts* (pp. 269). ACM Press.

Pareto Analysis. (n.d.). MSH and Unicef (Management Sciences for Health and the United Nations Children's Fund. Retrieved July 29, 2003, from http://erc.msh.org/quality/pstools/pspareto.cfm.

Pareto Analysis: Selecting the Most Important Changes To Make (n.d.). Mind Tools. Retrieved July 29, 2003, from http://www.mindtools.com/pages/article/newTED_01.htm.

Patton, M. Q. (1986). *Utilization-focused evaluation* (2nd ed.). Beverly Hills, CS: Sage.

Patton, M. Q. (1990). Qualitative evaluation and research methods (2nd ed.). Sage.

Payne, S. J., Sime, M. E., & Green, T. R. G. (1984). Perceptual structure cueing in a simple command language. *International Journal of Man-Machine Studies*, *21*, 19-29.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*(3), 295-341.

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM, 38*, 33-44.

Petre, M. (1996). Programming paradigms and culture: implications of expert practice. In M. Woodman (Ed.), *Programming language choice: Practice and experience* (pp.29-44). International Thomson Computer Press.

Pirolli, P. & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*, 240-272.

Polson, O., Lewis, C., Rieman, J., & Wharlton, C. (1992). Cognitive Walkthroughs: A method for theory-based evaluation of user-interfaces. *International Journal of Man-Machine Studies, 36*, 741-773.

Pong, M. C. & Ng, N. (1983). PIGS – A system for programming with interactive graphical support. *Software Practice and Experience, 13*(9), 847-855.

Programming languages have the usability of a. (n.d.). Retrieved December 28, 2002, from http://www.kuro5hin.org/poll/988809288_gWmMlSrs.

Ramsey, H. R., Atwood, M. E., & van Doren, J. R. (1983). Flowcharts versus program design languages: an experimental comparison. *Communications of the ACM, 26*(6), 445-449.

Ravden, S. J. & Johnson, G. I. (1989). *Evaluating Usability of Human-Computer Interfaces: A Practical Method.* Chichester: Ellis Horwood Limited.

Reese, W. L. (1980). *Dictionary of philosophy and religion.* Atlantic Highlands, NJ: Humanities.

Reiss, S. P. (1984). Graphical program development with PECAN program development systems. *ACM SIGPLAN Notices, 19*(5), 30-41.

Rieman, J. (1993). The diary study: A workplace-oriented research tool to guide laboratory efforts collecting user-information for system design. In *Proceedings of INTERCHI'93 Conference on Human Factors in Computing Systems* (pp. 321-326). ACM Press.

Rist, R. S. (1986). Plans in programming: definition, demonstration, and development. In E. Soloway & S. Iyengar (Eds.), *Proceedings of the Empirical Studies of Programmers: First Workshop* (pp.28-47). Norwood, NJ: Ablex.

Rothon, N. M. (1979). Design structure diagrams – a new standard in flow diagrams. *Computer Bulletin, 19*, 4-6.

Rubinstein, R. & Hersh, H. (1984). *The Human Factor.* Digital Press.

Samurçay, R. (1990). Understanding the cognitive difficulties of novice programmers: A didactic approach. In P. Falzon (Ed.), *Cognitive ergonomics: Understanding, learning and designing human-computer interaction* (pp.187-198). London: Academic Press.

Scanlan, D. A. (1989). Structured flowcharts outperformed pseudocode: An experimental comparison. *IEEE Software, 6*, 28-36.

Scrymarch (2001, May 4). Programming languages are human computer interfaces. Retrieved December 28, 2002, from http://www.kuro5hin.org/story/2001/5/2/92758/26768.

Sellen, A. J. (1994). Detection of everyday errors. *Applied Psychology: An International Review* (Special issue on error detection), *43*(4), 475-498.

Shackle, B. (1991). Context, framework, definition, design and evaluation. In B. Shackel & S. Richardson (Eds.) *Human Factors for Informatics Usability* (pp.21-38). Cambridge: Cambridge University Press.

Shneiderman, B. (1992). Designing the user interface: Strategies for effective human-computer interaction (2nd ed.). Addison-Wesley.

Shneiderman, B. & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, *8*(3), 219-238.

Shu, N. C. (1992). *Visual Programming.* New York: Van Nostrand Reinhold.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. (1977a). Structuring the programmer's task. *Journal of Occupational Psychology*, *50*, 205-216.

Sime, M. E., Green, T. R. G. & Guest, D. J. (1977b). Scope marking in computer conditionals-a psychological evaluation. *International Journal of Man-Machine Studies*, *9*, 107-118.

Sinha, A. P. & Vessey, I. (1992). Cognitive fit: an empirical study of recursion and iteration. *IEEE Transactions on Software Engineering*, *18*(5), 368-379.

Sinha, R., Hearst, M. Ivory, M., & Draisin, M. (2001). Content or graphics? An empirical analysis of criteria for award-winning websites. *Proceedings of the 7th Conference on Human Factors and the Web*, Retrieved February 22, 2002 from http://www.optavia.com/hfweb/7thconferenceproceedings.zip/

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE10*(5), 595-609.

Soloway, E., Bonar, J., & Ehrlich, K. (1983a). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, *26*(11), 853-860.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983b). What do novice know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in human-computer interactions* (pp.27-54). Norwood, NJ: Ablex.

Statistical thinking tools. (n.d.). Bob Luttman, Robert Luttman & Associates. Retrieved May 22, 2003, from http://www.robertluttman.com/Week5/page10.htm.

Steinman, S. & Carver, K. (1995). *Visual programming with Prograph CPX.* Manning Publications.

Suns usability labs and services. (n.d.). Retrieved May 21, from http://www.sun.com/usability/

Taylor, R. P., Cunniff, N. & Uchiyama M. (1986). Learning, research, and the graphical representation of programming. In *Proceedings of 1986 Fall Joint Computer Conference* (pp. 56-63).

Treisman, A. (1988). Features and objects: The fourteenth Barlett Memorial Lecture. *Quarterly Journal of Experimental Psychology: Human Experimental Psychology, 40A*, 210-237.

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

We have over 25 labs on the Redmonds campus. (n.d.). Retrieved May 21, 2003, from http://www.microsoft.com/usability/tour.htm.

Welty, C. & Stemple, D. (1981). Human factors comparison of a procedural and a non-procedural query language. *ACM Transactions on Database Systems, 6*(4), 626-649.

What is the Microsoft Usability Group all about? (n.d.). Retreived May 21, 2003 from http://www.microsoft.com/usability/faq.htm.

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing, 8*, 109-142.

Whitley, K. N. (2000). *Empirical research of visual programming languages: An experiment testing the comprehensibility of LabVIEW.* Unpublished Ph.D. thesis. Computer Science Department, Vanderbilt University, Nashville, TN.

Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies, 25*, 697-709.

Wiedenbeck, S. & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies, 51*, 71-87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11*, 255-282.

Wilde, N. P. (1996). Using Cognitive Dimensions in the classroom as a discussion tool for visual language design, *Proceedings of CHI'96 Conference on Human Factors in Computing Systems.* Retrieved January 16, 2002 from http://www.acm.org/sigchi/chi96/proceedings/shorpap/Wilde/wn_txt.htm.

Winn, W. (1993). An account of how readers search for information in diagrams. *Contemporary Educational Psychology, 18*, 162-185.

Witty, R. W. (1977). Dimensional flowcharts. *Software-Practice and Experience, 7*, 553-584.

Wright, P. & Reid, F. (1973). Written information: Some alternatives to prose for expressing the outcome of complex contingencies, *Journal of Applied Psychology, 57*(2), 160-166.

Yang, S., Burnett, M., DeKoven, E. & Zloof, M. (1995). *Representation design benchmarks: A design-time aid for VPL navigable static representations* (Technical report: TR 95-60-3). Oregon State University, Corvallis, OR.

Yourdon, E. (1989). *Modern structured analysis.* Englewood Cliffs, NJ: Prentice-Hall International.