Appeared in *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group*, A. F. Blackwell and E. Bilotta, Eds. Corigliano Calabro, Italy: Edizioni Memoria, April 10-13 2000, pp. 193-205.

The Influence of the Psychology of Programming on a Language Design: Project Status Report

John F. Pane and Brad A. Myers

Computer Science Department and
Human-Computer Interaction Institute
Carnegie Mellon University
pane+ppig2000@cs.cmu.edu

Keywords: POP-I.B barriers to programming, POP-II.A. novice programmers, POP-III.C all cognitive dimensions, POP-III.B new language.

Abstract

Research in Psychology of Programming (PoP) and related fields over the past thirty years has identified many important usability issues for programming languages and tools. However, when new programming languages are designed these findings do not seem to have much impact, so popular modern languages continue to exhibit many of the same old problems. This paper reviews the progress of an ongoing project to elevate the influence of PoP on the design of a new programming language. In the context of designing a new programming language for children, we cataloged and interpreted the prior work, performed new studies where questions remained unanswered, and have focused on usability throughout the design. In addition to producing a system that is easier to learn and use than existing systems, we hope to exemplify a process that could be adopted by other language designers to improve the usability of their systems.

Introduction

Our research is based on a recognition that the capability to program – to customize the behavior of computers – is a generally useful skill that can benefit many diverse people. Indeed, many people who are not

trained to be programmers would like to have this capability. However programming is well known to be a very difficult activity, especially for beginners. Some of this difficulty is intrinsic to programming, but some part of it can be relieved by careful attention to usability during the design of programming languages and tools. The goal of our research is to combine the specific knowledge that has been learned about the Psychology of Programming (PoP) with the general techniques and principles of Human Computer Interaction (HCI) to guide the design of new programming systems.

This paper summarizes our work to date on building a new programming system for children with a focus on usability. It begins with a review of several of the more prominent issues reported by prior research on beginner programmers, and describes some open questions that led us to conduct a pair of new studies to examine how children and other non-programmers naturally express problem solutions. These studies exposed some of the ways that current programming languages force people to express their solutions unnaturally. Informed by these results, we sketch a preliminary system design that addresses many of these issues. We note that the features of this design elevate the importance of query specification, a well-known area of difficulty for beginners. We describe a third study that characterized some of the problems users have with boolean expressions and tested some alternatives for query specification. Our new tabular query design performed significantly better than boolean expressions, and will be incorporated into the programming system. We are now finalizing the language design and implementing it, and will continue to perform additional studies with children to test its usability.

Historical Emphasis of Programming Language Research

As early as the mid-1980's, it was observed that programming language research and funding emphasized technical aspects of the domain and neglected psychological aspects:

> "Millions for compilers, but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. Yet we do not even have an enumeration of all of the psychological functions programming languages serve for the user. Of course, there is lots of programming language design, but it comes from computer scientists. And though technical papers on languages

> contain many appeals to ease of use and learning, they patently contain almost no psychological evidence nor any appeal to psychological science…The human and computer parts of programming languages have developed in radical asymmetry" (Newell & Card, 1985).

Since that time, substantial gains have been made in the PoP field and related areas, but this progress has been somewhat isolated from the technical research. New languages are usually spawned from technical demands and innovations, and gain their critical mass from adoption by the technical community. If PoP research is going to influence the designs of new languages, it must become visible to the technical community, and be in a form that can be readily applied.

Historically, the criteria that have guided design decisions have been based on technical objectives. For example, designers may articulate goals to build programming languages that are scalable, efficient, reusable, provably correct, or that have mathematical elegance, etc. When confronted by design decisions that are not fixed by these criteria, the designer might select features to be similar to existing languages, or to be different where prior languages are judged to be unsatisfactory on a particular point. Other times, the designer may use intuition to guide design decisions.

We suggest that PoP and HCI criteria and techniques should be routinely included among the pool of guidelines and strategies that are used to drive language design. These new criteria can be weighted as appropriate for the objectives of a particular design. In our new language for children, we take the extreme position of elevating usability criteria above all other objectives. But even designs that place foremost emphasis on certain technical requirements can benefit from consideration of the usability criteria.

Recommendations from Prior Research

Our review of prior research on beginners yielded a wealth of information. There is even more research to consider when the scope is expanded beyond beginners. However, it is a challenging task to gather this literature from its diverse sources and organize it into a form that can be applied in a disciplined way to language design. It would be a great contribution for the PoP community to publish a definitive reference handbook, collecting and organizing the state of knowledge in this field. Our own collection of the research applicable to beginners appears in a technical report (Pane & Myers, 1996). While we intend to take into account all of the points therein as we design our language, there are too many to detail in this paper. It is worthwhile, however, to highlight some of the more influential research and the themes that are most prevalent.

First, the field of HCI has general principles or heuristics that apply to programming system design, such as to be consistent, keep it simple, speak the user's language, prevent errors, help the user get started, etc. (Nielsen, 1994). The *cognitive dimensions* framework provides an excellent set of more specific guidelines for assessing programming systems, such as closeness of mapping, viscosity, hidden dependencies, imposed guess-ahead and visibility, etc. (Green & Petre, 1996). However, it is difficult to optimize along all of these dimensions at once, because they are not independent. Improving a system on one measure may result in reduced performance on another. To guide us in making these tradeoffs it is useful to review studies of the target audience that identify the problems that are common.

In our review of studies of beginners, the most prominent problems seemed to fall along the following cognitive dimensions and HCI principles:

- Visibility. Memory overload is a problem for all programmers, but is particularly troublesome for beginners because they have not yet developed strategies to relieve their memory load (Anderson & Jeffries, 1985; Davies, 1993). The programming system should make information visible or readily accessible any time it is relevant.
- Closeness of mapping. Programming is a process of translating a mental plan into one that is compatible with the computer (Hoc & Nguyen-Xuan, 1990). The language should minimize the difficulty of this translation by providing high-level primitives that match the operators in the plan, including those that may be specific to a domain the programmer is addressing in the program. If the language does not provide these high-level operators, programmers are forced to compose low-level primitives to achieve their high-level goals. This synthesis has been called one of the greatest cognitive barriers to programming (Lewis & Olson, 1987).
- Speak the user's language. The language should avoid using words and symbols that are unfamiliar to users, or that have conflicting meanings in other domains. When users are not sure what to do they attempt to transfer knowledge from other domains that are more familiar, including natural language and mathematics (Hoc & Nguyen-Xuan, 1990). This will result in errors and confusion if the semantics of the programming language is not compatible with the semantics in these other domains.

In addition, researchers have noted that the *von Neumann* computational model is a stumbling block for beginners because it is not familiar and has no real-world counterpart (du Boulay, 1989; du Boulay,

O'Shea, & Monk, 1989). Usability could be improved by selecting a different model of computation based on a concrete real-world system that is familiar to the audience (Mayer, 1989; Smith, Cypher, & Spohrer, 1994). This permits users to infer how the programming system will work by consulting their existing knowledge and expectations about the modeled system.

There are also several language constructs or features that have been identified as especially troublesome. For example, looping control structures have been studied extensively because they are so difficult. Researchers have identified ways to improve performance by redesigning these structures to more closely match the mental plans of the users (Soloway, Bonar, & Ehrlich, 1989; Wu & Anderson, 1991). But it is also notable that languages sometimes require users to perform iteration in situations where aggregate operations can accomplish the task more easily (Miller, 1981). Another notoriously difficult task is the accurate specification of boolean expressions. This has been studied extensively because it is important in other areas besides programming, such as in database retrieval tasks. Researchers have observed that the common uses of the words *AND*, *OR*, and *NOT* in natural language lead to errors when these words are used to name the boolean operators in queries (Greene, Devlin, Cannata, & Gomez, 1990); and the intended scope of the *NOT* operator is ambiguous (McQuire & Eastman, 1995).

A somewhat controversial topic is whether to use a textual or a visual language. Many people argue that programming is difficult because it requires the precise use of a textual language, and that a system that eliminates language will be inherently easier to use (e.g. Smith et al., 1994). However, numerous researchers have demonstrated situations where textual languages outperform visual languages (e.g. Green & Petre, 1992). It seems that visual languages might be better for small tasks, but often break down for large tasks. The good news is that it is not always difficult to learn to program in a textual language. In fact, the most successful end-user programming system is the spreadsheet, which is text-based (Nardi, 1993).

Studying the Language and Structure used by Non-Programmers

Much of the prior work offers guidance about things to avoid in programming language design, but there are fewer prescriptions for good ways to implement the necessary features. Since a failure to provide closeness of mapping can explain why certain aspects of programming are difficult, we set out to study problem-solving by non-programmers. Our objective was to identify properties of these solutions that illuminate how the language can be designed to better match beginners' mental plans.

We designed a pair of studies to investigate the language and structure used by non-programmers in their solutions to programming problems, similar to studies conducted in the 1970's by Miller (1974; 1981). To begin, we enumerated a list of essential concepts that are necessary in programming, e.g. use of variables, boolean logic, arithmetic, iteration, etc. We then selected a set of tasks that evoke these concepts from two kinds of programs: an arcade game and a program that transforms and calculates tabular data. Because we wanted to capture the natural responses of the participants, we took care not to bias the language they used in their responses. To accomplish this, we displayed graphical depictions of the tasks along with terse captions. The participants were mostly fifth-graders from a local public school, of mixed race, gender and academic ability.

In examining the results, we saw interesting trends in the solutions. To ensure that we were not just finding what we were looking for, we developed a rating form to allow unbiased experts to characterize the solutions on various dimensions. This was done by providing a rating form to five independent analysts who are experienced programmers. The following results are based on these analysts' ratings.

The study found that the most prevalent style of programming suggested by the solutions was a rule-based or event-based style ("if pacman loses all his lives, it's game over"). The participants tended to perform operations in aggregate, rather than using iteration ("the monsters turn blue and run away"). A natural language style was used for arithmetic ("add 1000 to score"), but users who used mathematical expressions were more accurate. Objects were normally moving, and remembered their state ("if pacman hits a wall, he stops"). Operations seemed more consistent with list data structures rather than arrays. The participants did not construct complex data structures and traverse them, but instead performed queries to obtain the necessary information when needed.

Boolean expressions were avoided in many cases. Instead, the raters identified other ways the participants expressed the necessary logic, such as a mutually exclusive set of rules, or a general case followed by an exception. It was observed that AND was frequently used where disjunction ("or") was intended, and that NOT was treated with lower precedence than the other boolean operators, contrary to the precedence used in most programming languages. Pane, Ratanamahatana & Myers (2000) describes the full details of these studies.

Implications for Design

With the prior work and the results of these studies, we began to develop a design for our new programming system for children. The

details of this design are currently being refined and tested, but this section sketches out the general idea.

We have chosen a *meeting* as a concrete model for the computation. In this scenario, a program is represented as a group of independent *agents* seated around a *table*.[1] The agents represent encapsulated pieces of the program, and may work in cooperative or adversarial roles to accomplish tasks. Some of the agents are programmed by the programmer, while others may represent libraries of functionality that are provided by the system.

All program data is represented on *cards*, which have unique names and include an unlimited number of slots containing attribute-value pairs. Cards held in the hand of a particular agent are private to that agent, so they are invisible to the other agents. Cards that are placed on the table are public, and so they are visible and modifiable by all agents. These features provide mechanisms for data hiding and sharing. A special area in the center of the table is designated as the *board*, which represents the runtime screen of the program as seen by the end-user. Cards that are placed on the board are displayed in a special way at runtime, with only the contents of their *appearance* slot visible.

We have chosen a textual representation for the program to help ensure that the proposed language will scale gracefully as the user builds larger and more complex applications. Each agent has its own textual *agenda* in a thought-bubble above its head, representing its program. The agenda consists of a set of *event handlers*, which are actions to perform when an event occurs. The system dispatches events by placing *event cards* on the table. When an agent sees an event for which it has a handler, the *action* of that handler is invoked. As part of the action, any visible card can be viewed or changed, new cards can be created by drawing blank cards from the *new card pile* and filling in their slots, and existing cards can be deleted by moving them to the *discard pile*. The agent can delete the event card that invoked the action, preventing other agents from seeing it. If it does not do this, other agents that have an event handler for that event will have an opportunity to run before the system cleans up by removing the event. Several choices are available to determine which agent goes first when there are multiple candidates – this issue is yet to be decided.

The language will support queries for accessing data. The result of a query is a group of zero or more cards that can be operated on directly by the operators of the language. The programmer will not be required to write separate cases to handle the situations where zero objects are

---

[1] Note that the words used here for the components of the system were chosen for the audience of this paper. We are still working on the exact terminology to use in the final system.

returned, one object is returned, or multiple objects are returned. These features reflect our observations that users access data by performing queries instead of traversing data structures, and that they operate on groups of objects in aggregate.

The system will supply agents that provide high-level functionality to the programmer. For example, an *animation agent* may provide automatic animation of objects that are on the board, based on the values in slots such as position, velocity, etc. A *collision agent* may monitor the motions of objects on the board, and report collisions among them by placing collision events onto the table. A *timer agent* may provide services related to periodic or one-time notification of the elapse of time, by creating event cards that are visible to other agents. This mechanism for encapsulating high-level behavior into agents is also available to the programmer, enabling the creation of new abstractions that can be easily shared among programs.

Figure 1 contains a sample event handler for a simplified implementation of Pacman. This event handler will run after the system's collision agent has detected a collision and generated a collision event. The event handler responds by inspecting the event to see which objects collided. If the collision was between Pacman and a wall, the velocity slot on the card representing Pacman is set to zero, causing the animation agent to stop moving Pacman forward. If the collision was between Pacman and a powerpill, the system is queried for all of the monsters, and it returns a list of four monster cards. The handler then sets the color slots to blue on all of the monsters in aggregate using the same operator that was used earlier to set a single slot.

```
at the time that a Collision event appears
   if the Collision event contains Pacman then
      if the Collision event contains
         a Wall then
            set Pacman's Velocity to 0
         a Powerpill then
            with all of the cards that match Monster calling them M
               set M's Color to Blue
            end with
      end if
   end if
end at
```

Figure 1: Example of an event handler that responds to collisions between Pacman and a wall or a powerpill.

The textual language is verbose and uses a natural language style similar to HyperTalk (Goodman, 1987), but we have taken care to avoid

many of the consistency problems in that language (Thimbleby, Cockburn, & Jones, 1992). The programming environment will offer structure editing features, such as popup menus and command completion, to help users build syntactically correct programs and reduce typing. White space will not be significant, although the environment will try to indent the code appropriately. A small set of throw-away words (e.g. *the*, *a*, and *an*) will be ignored, so they can be used wherever needed to make the code more readable. Statements do not have to be terminated with punctuation such as semicolons, but control structures that can contain multiple statements must have end delimiters. These delimiters repeat the name of the control structure to provide extra cues for readability (Fitter & Green, 1979). A single conditional control structure offers the features of *if* statements and *case* statements in a unified way. Query results can be named temporarily with an identifier by using a *calling-it* clause, (as suggested by Miller, 1981), but using the pronoun *it* (as in HyperTalk) was rejected because it is too easy when editing to change the binding of this variable accidentally.

This design attempts to provide the essential features of a programming system in a way that responds to the research on beginner programmers. The meeting model offers a concrete and familiar metaphor for computation, unlike the *von Neumann* model. Cards provide visibility and concreteness for variables and data objects, which are represented in an abstract and invisible way in most other languages. The model supports the event-based style of programming that was observed to be common in non-programmers' problem solving. This system encapsulates state with objects, and the system-provided agents can provide autonomy, but objects do not encapsulate code or support inheritance because these features seem to be inappropriate for beginners.

Boolean Queries Revisited

The queries and aggregate operations featured in the proposed system elevate the importance of a usable query language. The prior work does not offer a ready solution, so we conducted a further study to investigate alternative designs for the query mechanism. We included several of the alternative formulations that were observed on the earlier studies, such as using a general case followed by an exception, to see whether they were more accurate than traditional boolean expressions. In addition, because prior research suggests that non-textual query languages may be more effective than textual syntaxes (e.g. Young & Shneiderman, 1993), the study compared these textual alternatives against a proposed new query language that uses tabular forms that we could integrate easily into our textual language.

Since our new programming language will represent data on cards containing attribute-value pairs, we designed the query form to also use a

card metaphor. For the purposes of this study, we simplified the forms by leaving out the attribute names, and limiting the number of terms to three. We called these match forms (see Figure 2). Criteria are placed into the slots, one term per slot. All of the terms on a single form implicitly form a conjunction. Negation is specified by prefacing a term with the NOT operator. Disjunction is specified by including an additional match form adjacent to the first one.

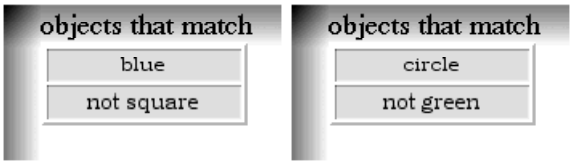

Figure 2. Match forms expressing the query: (blue and not square) or (circle and not green)

This two-dimensional layout is similar to the grid of tiles described by Anick et al. (1990) – one dimension implements intersection and the other implements union. However, match forms provide cues to help users remember which operator uses each dimension, such as the text in the form heading and the visual grouping. In addition, the scope of the NOT operator is made explicit by confining it to a single term. This proposed query language can express arbitrarily complex queries, although some queries have to be formulated in a less concise way than pure boolean expressions would allow. To relieve this somewhat, the forms in our proposed language will also allow an entire form to be negated ("objects that do not match ..."), but that feature is not used in this study.

Participants were presented with a series of problems that tested their ability to interpret queries that we provided, and their ability to generate queries to achieve a desired result. In the interpretation tasks, they were shown a query and a grid of nine colored shapes, and were instructed to checkmark the shapes that would be selected by the query. In the generation task, they were presented with the same grid of shapes, some of which were already checkmarked, and were instructed to write a query that would result in the desired selection. In order to avoid confounding the results, presentation order of the various problems was counterbalanced, except that all of the query generation tasks were completed before any of the query interpretation tasks were presented.

Thirteen children and twenty adults participated in this study. Because there were no significant differences between the children and the adults, the following results are reported for the group as a whole.

In summary, none of the alternative textual formulations for queries were significantly better than boolean expressions. However, we did observe significant trends in the way various queries are interpreted that illustrate some of the confusions that people have with boolean expressions. Interpretation of AND varied according to context: 85% interpreted "select the objects that match blue and circle" as a boolean conjunction while only 36% interpreted "select the objects that match blue and the objects that match circle" as boolean conjunction ($p<.0001$). In the latter query, 55% interpreted the query as disjunction ("or").

There was an interesting reversal in the precedence attributed to the NOT operator. In the expression "select the objects that match not red and square," 64% interpreted NOT with high precedence, as if the expression were parenthesized as "(not red) and square," and 9% interpreted it with low precedence ($p<.001$). However, in the expression "select the objects that match not triangle or green," only 9% interpreted NOT with high precedence, and 67% interpreted it with low precedence ($p<.001$). When we supplied parenthesis to try to override this tendency in the expression "select the objects that match (not circle) or blue", 39% of the participants ignored the parenthesis and still gave the NOT lower precedence than the OR, while only 12% gave it high precedence ($p<.05$).

The results with tabular query forms were encouraging. Participants performed about equally well on query forms and boolean expressions when they were interpreting queries, but performed significantly better ($p<.0001$) generating queries with match forms (94% correct) than with boolean expressions (85% correct).

Our conclusions from this study are that the word AND should be avoided in programming languages because it is misinterpreted so often. Also, it is dangerous to rely on implicit operator precedence rules, and use of parenthesis for expression grouping is unreliable. However, tabular query forms offer promise in improving the accuracy of queries, and we will use them in our new system. Full details of this study are reported in Pane & Myers (2000).

Current and Future Work

We are currently refining the details of our new computational model and testing whether users will understand it and know what to do in order to accomplish their goals. The system we are implementing will include a complete interactive programming environment, with features such as structure editing and word completion to facilitate program entry, integrated debugging tools, etc. We will test the final product in comparison with other programming systems for children, and hope to demonstrate that our system is easier to learn and use.

In addition to producing this new system for children that contains a new model for computation and other features suggested by PoP and HCI

research, we plan to apply this same process to developing new systems for other audiences and domains. Hopefully this work will inspire other language designers to give a more prominent role to PoP and HCI research during the design process.

Acknowledgements

This research is funded in part by the National Science Foundation under Grant No. IRI-9900452. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

Anderson, J. R., & Jeffries, R. (1985). Novice LISP Errors: Undetected Losses of Information from Working Memory. *Human-Computer Interaction, 1*, 107-131.

Anick, P. G., Brennan, J. D., Flynn, R. A., Hanssen, D. R., Alvey, B., & Robbins, J. M. (1990). A Direct Manipulation Interface for Boolean Information Retrieval via Natural Language Query, *Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 135-150). Brussels, Belgium.

Davies, S. P. (1993). Externalising Information During Coding Activities: Effects of Expertise, Environment and Task. In C. R. Cook, J. C. Scholtz, & J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop* (pp. 42-61). Palo Alto, CA: Ablex Publishing Corporation.

du Boulay, B. (1989). Some Difficulties of Learning to Program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 283-299). Hillsdale, NJ: Lawrence Erlbaum Associates.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 431-446). Hillsdale, NJ: Lawrence Erlbaum Associates.

Fitter, M. J., & Green, T. R. G. (1979). When Do Diagrams Make Good Computer Languages? *International Journal of Man-Machine Studies, 11*, 235-261.

Goodman, D. (1987). *The Complete HyperCard Handbook*. New York: Bantam Books.

Green, T. R. G., & Petre, M. (1992). When Visual Programs are Harder to Read than Textual Programs. In G. C. van der Veer, M. J. Tauber, S. Bagnarola, & M. Antavolits (Eds.), *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)* . Rome: CUD.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing, 7*(2), 131-174.

Greene, S. L., Devlin, S. J., Cannata, P. E., & Gomez, L. M. (1990). No IFs, ANDs, or ORs: A Study of Database Querying. *International Journal of Man-Machine Studies, 32*(3), 303-326.

Hoc, J.-M., & Nguyen-Xuan, A. (1990). Language Semantics, Mental Models and Analogy. In J.-M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 139-156). London: Academic Press.

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.

Mayer, R. E. (1989). The Psychology of How Novices Learn Computer Programming. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 129-159). Hillsdale, NJ: Lawrence Erlbaum Associates.

McQuire, A., & Eastman, C. M. (1995). Ambiguity of Negation in Natural Language Queries, *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 373).

Miller, L. A. (1974). Programming by Non-Programmers. *International Journal of Man-Machine Studies, 6*(2), 237-260.

Miller, L. A. (1981). Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Systems Journal, 20*(2), 184-215.

Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: The MIT Press.

Newell, A., & Card, S. K. (1985). The Prospects for Psychological Science in Human-Computer Interaction. *Human-Computer Interaction, 1*(3), 209-242.

Nielsen, J. (1994). Heuristic Evaluation. In J. Nielsen & R. L. Mack (Eds.), *Usability Inspection Methods* (pp. 25-62). New York: John Wiley & Sons.

Pane, J. F., & Myers, B. A. (1996). *Usability Issues in the Design of Novice Programming Systems* (School of Computer Science Technical Report CMU-CS-96-132). Pittsburgh, PA: Carnegie Mellon University.

Pane, J. F., & Myers, B. A. (2000). Tabular and Textual Methods for Selecting Objects from a Group. *submitted for publication*, http://www.cs.cmu.edu/~pane/Study3.html.

Pane, J. F., Ratanamahatana, C. A., & Myers, B. A. (2000). Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal on Human-Computer Studies, to appear*.

Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming Agents Without a Programming Language. *Communications of the ACM, 37*(7), 54-67.

Soloway, E., Bonar, J., & Ehrlich, K. (1989). Cognitive Strategies and Looping Constructs: An Empirical Study. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 191-207). Hillsdale, NJ: Lawrence Erlbaum Associates.

Thimbleby, H., Cockburn, A., & Jones, S. (1992). HyperCard: An Object-Oriented Disappointment. In P. Gray & R. Took (Eds.), *Building Interactive Systems: Architectures and Tools* (pp. 35-55). New York: Springer-Verlag.

Wu, Q., & Anderson, J. R. (1991). Strategy Selection and Change in Pascal Programming. In J. r. Koenemann-Belliveau, T. G. Moher, & S. P. Robertson (Eds.), *Empirical Studies of Programming: Fourth Workshop* (pp. 227-238). New Brunswick, NJ: Ablex Publishing Corporation.

Young, D., & Shneiderman, B. (1993). A Graphical Filter/Flow Representation of Boolean Queries: A Prototype Implementation and Evaluation. *Journal of American Society for Information Science, 44*(6), 327-339.