A Programming System for Children that is Designed for Usability (Chapter 1)

This document contains only Chapter 1 of the thesis.
For citation, please refer to the full thesis document, which is available at:
http://www.cs.cmu.edu/~pane/thesis

John F. Pane
CMU-CS-02-127
May 3, 2002

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Brad A. Myers (co-chair)
David Garlan (co-chair)
Albert Corbett
James Morris
Clayton Lewis, University of Colorado

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Also appears as: CMU-HCII-02-101

Copyright © 2002 John F. Pane

This research was sponsored 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 author and do not necessarily reflect those of the National Science Foundation.

Keywords: Natural Programming, HANDS, End-User Programming, Psychology of Programming, Empirical Studies of Programmers, Educational Software, Children, User Interface Design, Programming Environments, Programming Language Design, Usability, Human-Computer Interaction.

Abstract

A programming system is the user interface between the programmer and the computer. Programming is a notoriously difficult activity, and some of this difficulty can be attributed to the user interface as opposed to other factors. Historically, the designs of programming languages and tools have not emphasized usability.

This thesis describes a new process for designing programming systems where HCI knowledge, principles and methods play an important role in all design decisions. The process began with an exhaustive review of three decades of research and observations about the difficulties encountered by beginner programmers. This material was catalogued and organized for this project as well as for the benefit of other future language designers. Where questions remained unanswered, new studies were designed and conducted, to examine how beginners naturally think about and express problem solutions. These studies revealed ways that current popular programming languages fail to support the natural abilities of beginners.

All of this information was then used to design HANDS, a new programming system for children. HANDS is an event-based system featuring a concrete model for computation based on concepts that are familiar to non-programmers. HANDS provides queries and aggregate operations to match the way non-programmers express problem solutions, and includes domain-specific features to facilitate the creation of interactive animations and simulations. In user tests, children using the HANDS system performed significantly better than children using a version of the system that lacked several of these features. This is evidence that the process described here had a positive impact on the design of HANDS, and could have a similar impact on other new programming language designs.

The contributions of this thesis include a survey of the knowledge about beginner programmers that is organized for programming system designers, empirical evidence about how non-programmers express problem solutions, the HANDS programming system for children, a new model of computation that is concrete and based on familiar concepts, an evaluation of the effectiveness of key features of HANDS, and a case study of a new user-centered design process for creating programming systems.

Acknowledgements

I would like to extend my heartfelt appreciation to Brad Myers for his insight and guidance throughout my Ph.D. work. I am very grateful for the many hours that he spent discussing and critiquing my work.

I am also very thankful for the feedback and support of my co-advisor, David Garlan, and the other members of my committee: Jim Morris, Clayton Lewis and Albert Corbett. Albert was especially generous in the time he spent helping me to design the user studies and analyze the results.

Many other faculty at CMU and elsewhere gave me valuable feedback and suggestions along the way. In particular, I would like to thank Bonnie John, Ken Koedinger, Wayne Gray, Margaret Burnett, Alan Blackwell, and Thomas Green.

I am especially grateful to Bonnie John, Dana Scott and Phil Miller, who were influential in my decision to become a Ph.D. student, and who helped me gain admission to the program.

I would like to thank the undergraduate and master's students who helped me develop the ideas in HANDS and who worked on the user studies: Leah Miller, Chotirat "Ann" Ratanamahatana, John Chang, Gabe Brisson, Luis Cota, and Ruben Carbonnell. Thanks to Joonhwan Lee for creating the graphics in the HANDS system. Thanks to Rob Miller for contributing his code for multi-level undo in the text editor.

Many thanks to Bernita Myers for acting as liaison to the East Hills Elementary school. Thanks to Mr. Niklos, the principal, as well as the teachers who allowed us to work in their classrooms: Carol Beavers and John Meighan. Also, thanks to Laurie Heinreicher at Winchester-Thurston school. Thanks to Michael Pane for his assistance in pilot testing the user study evaluating HANDS, and Melody Mostow for doing this and also starring in the HANDS video. Thanks to Ryan and Reid Myers, who helped us recruit volunteers for one of the studies. And, special thanks to Ryan for his insightful comparison of HANDS with Stagecast. And of course, thanks to all of the participants in my studies.

Thanks to Gary Perlman for working with me to develop and evaluate the search interface for the HCI Bibliography.

Thanks to many additional friends and fellow students who have helped me in various ways, especially Neil Heffernan, Chuck Rosenberg, Laurie Hiyakumoto, Herb Derby, Eugene Ng, Adam Berger, Matt Zekauskas, Maria Ebling, Chris Long, and David Eckhardt. A special thanks to Drew Morgan whose friendship and counsel was essential to my ability to make it through this project.

Especially, I would like to express my gratitude to my family for their support and encouragement. Most of all, thanks to my wife Barbara, who has given me her patient loving support throughout. Without this I may have never made it. I hope the rest of our lives will give me sufficient opportunity to reciprocate. Finally, thanks to Lorenzo, our next big project and one of the compelling motivations to finish this one.

Contents

Abstract **iii**

Acknowledgements **v**

CHAPTER 1 *Introduction* ***1***

Historical Context **2**

A User Centered Design Process for Programming Systems **3**

Motivation **4**

Thesis Statement **4**

Target Audience and Domain **5**

Understanding the Target Audience **5**

General Design Principles ***5***

Observations about Existing Programming Languages ***7***

Naturalness ***9***

Studies of Naturalness in Problem Solving ***10***

Study of Methods to Specify Queries ***11***

Model of Computation ***12***

Visual vs. Textual ***13***

The HANDS Programming System Design **13**
Computational Model ***14***
Programming Style and Model of Execution ***15***
Aggregate Operations ***15***
Queries ***15***
Domain-Specific Support ***16***
Evaluation **16**
Contributions **17**
Overview of Thesis **18**

CHAPTER 2 *Related Work* ***19***

Usability Issues in Programming Systems for Beginners **19**
Systems for Beginners and Children **19**
The Logo Family ***19***
Boxer ***20***
ToonTalk ***20***
AgentSheets ***21***
Stagecast ***21***
SmallTalk and eToys ***23***
Alice ***24***
Rehearsal World ***25***
Karel the Robot ***25***
GRAIL ***26***
HyperTalk ***26***
AppleScript ***26***
SK8Script ***26***
Chart 'n' Art ***27***
cT ***27***
LabView ***27***
Forms/3 ***27***
Visual Basic ***27***
Java and C# ***28***
MacGnome ***28***
Programming by Demonstration ***29***
Hank ***29***

CHAPTER 3 *The Language and Structure in Problem Solutions Written by Non-Programmers* **31**

Comparison to Lance Miller's Studies **32**
Overview of the Studies **33**
Study One **34**
Participants **34**
Materials **35**
Procedure **36**
Content Analysis **36**
Results **37**
Overall Structure **40**
Keywords **41**
Control Structures **42**
Computation **44**
Discussion **46**
Study Two **46**
Participants **46**
Materials **47**
Procedure **48**
Content Analysis **48**
Results **49**
Keywords **49**
Control Structures **52**
Computation **53**
Discussion of Results **58**
Programming Style **58**
Summary of These Studies **65**

CHAPTER 4 *Methods for Expressing Queries* **67**

Overview **67**
Prior work on Boolean Queries **70**
Design alternatives for Boolean queries **70**
Tabular query forms **71**
Hypotheses **72**
AND vs. nested IF **72**
NOT vs. Unless **72**

Location of Unless ***72***
Context-dependent interpretation of AND ***73***
Verbose AND vs. OR ***73***
Operator precedence of NOT ***73***
Parentheses for expression grouping ***74***
Tabular vs. textual ***74***

Method **74**
Participants ***75***
Materials ***76***
Procedure ***76***

Results **78**

Discussion **80**
Textual query variations ***81***
Match forms vs. text ***82***

Summary **83**

Application of Results **84**

CHAPTER 5 *The HANDS System* *87*

Motivating Factors in the HANDS Design **87**

Representation of the Program **89**
Cards for Data Storage ***89***
Computation is Performed by Handy ***94***

Programming Style and Model of Execution in HANDS **96**
Structure of Event Handlers ***97***
Event Dispatch ***97***
The Events ***98***
Event Patterns ***99***
Event Cards ***101***

Data Types **101**

Numeric Values and Calculations **102**

Language Syntax **103**
Natural-Language Style ***103***
Plurals ***103***
Control Structure Terminators ***104***
Statement Terminators ***104***
Parentheses Are Required to Indicate Precedence Explicitly ***104***
List Syntax ***105***

Consistency Between Values on Cards and in Program Code ***105***
Comments, Indenting, and White Space ***105***
Choices for Keywords and Special Identifiers ***106***
Statements **106**
Operations on Cards ***107***
Operations on Card Properties ***107***
Output Statements ***109***
Other Statements ***109***
Expressions **109**
Relational Operators ***110***
Boolean Operators ***110***
Card Existence Predicate ***111***
Mathematical Operators ***111***
Random ***111***
Expression for Getting Input from User ***112***
Aggregate Operations **112**
Queries **113**
Queries and Aggregates in Combination **114**
List Operators **114**
Loop and Conditional Control Structures **119**
Iteration Control Structure ***119***
Conditional Control Structure ***121***
Domain-Specific Support **123**
Graphical Objects ***123***
Animation ***124***
Mouse Click Detection ***127***
Collision Detection ***127***
Coordinate System ***129***
Programming Environment **129**
System-wide Menu Commands ***129***
Event Browser ***131***
Testing Window ***136***
Cards Window ***137***
Handy's Hand ***137***
Runtime Errors **138**
Implementation Details **141**
HANDS Runtime Implementation ***141***
Format for Saved Files ***143***

Sample Program 143
Importing Components 145
Summary 146

CHAPTER 6 *Evaluation* ***147***

User Study **148**
- *Queries and the Alternative* ***149***
- *Aggregate Operators and the Alternative* ***150***
- *Visibility of Data and the Alternative* ***150***

The Study **152**
- *Participants* ***152***
- *Materials* ***152***
- *Procedure* ***153***
- *Results* ***153***
- *Informal Observations* ***155***
- *Summary of Study* ***155***

Example Programs **156**
- *Breakout Game* ***156***
- *Simulation of the Ideal Gas Law* ***156***
- *Towers of Hanoi* ***157***
- *Computing Prime Numbers* ***158***

Comparison with Another System **159**
Some Weaknesses of HANDS **161**
Range of Capabilities **161**
Programming Strategies **162**
Evaluation of Earlier Design Ideas **163**
Some Criticisms of HANDS **165**
Summary of Evaluation **167**

CHAPTER 7 *Future Work* ***169***

Further Evaluation and User Testing **169**
Ideas for Extending HANDS **170**
- *Modularity and Encapsulation* ***170***
- *Multiple Agents* ***171***

Graphics Primitives ***172***
Improvements to Collision Detection and Animation ***172***
Timers ***173***
Match Forms ***173***
Widget Library ***173***
Dealing with Large Numbers of Cards ***174***
Editing and Debugging Support ***174***
HANDS as a Complete Package for Teachers and Students ***175***
Applications of Results to Other Areas **175**
Model of Computation ***175***
Export Features to Other Languages ***175***
Influence Design Process for Future Languages and Domains ***176***
Applications of Match Forms ***176***

CHAPTER 8 *Conclusion* ***177***

Contributions **177**
Design Process ***177***
HANDS ***178***
Tabular Method for Expressing Boolean Queries ***178***
User Studies ***178***
Survey of Prior Work ***179***
Closing Remarks **179**

CHAPTER 9 *References* ***181***

APPENDIX A *Language Syntax Chart* ***193***

APPENDIX B *Example Programs* ***215***

Breakout **216**
Ideal Gas Law Simulation **222**
Towers of Hanoi **228**
Extension to Towers of Hanoi ***229***
Primes Sieve **230**
Compass **230**

Boundaries 231
Trap Door 232

APPENDIX C *Background Research* ***233***

APPENDIX D *Materials from Study 1* ***321***

APPENDIX E *Materials from Study 2* ***341***

APPENDIX F *Materials from Study 3* ***381***

APPENDIX G *Materials from Study 4* ***415***

CHAPTER 1 Introduction

Only a very small proportion of users can program their computers. However, most could benefit in some way from this powerful capability, whether to customize and interconnect their existing applications or to create new ones. As with writing, “the significance of programming derives not only from the carefully crafted works of a few professionals, but also from the casual jottings of ordinary people” [diSessa 1986, p. 859]. For ordinary people, understandability, familiarity, ease of performing small tasks, and user interface are more important features in a programming system than technical objectives such as mathematical elegance, efficiency, verifiability, or uniformity.

Many of the people who try to learn to program are quickly discouraged because it is very difficult. In fact, it is even challenging for more experienced people who have received formal training. Why is programming so difficult? Part of the problem is that it requires problem solving skills and great precision, but this does not fully explain the difficulty. Even when a person can envision a viable detailed solution to a programming problem, it is often very hard to express the solution correctly in the form required by the computer. This is a user-interface problem that has long been recognized but neglected.

1.1 Historical Context

In 1971, Gerald Weinberg published *The Psychology of Computer Programming*, with the stated goal to trigger a new field that studies computer programming as a human activity [Weinberg 1971]. At the time, there was little scientific literature about the human aspect of programming, and most of it appeared in technical reports and other obscure publications. The field began to grow quickly after Allen Newell addressed the third ACM *CHI Conference on Human Factors in Computing Systems*, and later published his comments in an article with Stuart Card:

> 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. [Newell 1985, p. 212]

Soon two workshop series were started, which have become focal points for research in the usability of programming languages: the *Psychology of Programming Interest Group* (PPIG) explores the cognitive aspects of computer programming; and the *Empirical Studies of Programmers* (ESP) group focuses on empirical studies of beginners and experts.

Over the past three decades, many researchers have worked to understand the cognitive demands of programming and the sources of difficulty in existing programming languages and tools. In addition to the proceedings of the PPIG and ESP workshop series, relevant work has appeared in the *International Journal of Human-Computer Studies* (formerly *International Journal of Man-Machine Studies*), the proceedings of the ACM *CHI* conference and the IEEE *Human-Centric Computing* (formerly *Visual Languages*) conference,

and the books *Studying the Novice Programmer* [Soloway 1989b], *Psychology of Programming* [Hoc 1990a], and *Software Design: Cognitive Aspects* [Détienne 2001].

1.2 A User Centered Design Process for Programming Systems

It is disappointing that the knowledge gathered over the past thirty years has had so little influence on the designs of new programming systems (in this document, the term programming system is used to encompass the programming language as well as the tools for viewing, editing, debugging and running programs). In order to help remedy this, I have organized the prior work that studied beginner programmers so that it might be readily included among the guidelines and strategies that are used by future programming system designers. Generally, language designers have focused on technical goals for their systems, such as to build systems that are scalable, efficient, reusable, provably correct, or that have mathematical elegance. When they face a design decision that is not determined by these criteria, they usually choose a solution that is similar to existing languages or one that appeals to their intuition. Usability has rarely been adopted as a formal objective.

I believe that usability should always be included among the criteria that are considered during the design of programming systems. Depending on the constraints of a particular project and target audience, usability may be given more or less weight. However, it is always worth considering for at least those decisions that are not already determined by other design criteria

In this thesis, I exemplify a new design process for programming systems, where usability is treated as a first-class objective:

1. *Identify the target audience* and the domain, that is, the group of people who will be using the system and the kinds of problems they will be working on.
2. *Understand the target audience*, both the problems they encounter and the existing recommendations on how to support their work. This includes an awareness of general HCI principles as well as prior work in the psychology of programming and empirical studies. When issues or questions arise that are not answered by the prior work, conduct new studies to examine them.
3. *Design the new system* based on this information.

4. *Evaluate the system* to measure its success, and understand any new problems that the users have. If necessary, redesign the system based on this evaluation, and then re-evaluate it.

In this design process, all of the prior knowledge about the human aspects of programming are considered, and the strategy for addressing any unanswered questions is to conduct user studies to obtain design guidance and to assess prototypes. For my new programming system for children, I adopted an extreme position by giving usability precedence over other objectives.

While my focus has been on beginner programmers, I believe this approach also applies to experts, and that it can have positive impacts on training and productivity as well as the reliability of professional software systems. Improving the programming systems used by experts will also affect beginners, because although these systems may not be the best choices for learning to program, they are often chosen because they are widely available and familiar to mentors. Everyone would benefit if these programming languages and tools were more usable.

1.3 Motivation

The goal of this thesis is to *enable* more beginners to learn to program for their personal purposes, with minimal training. There is no explicit goal to teach any particular computer science concepts, such as recursion, unless the concept is essential to the users achieving their goals. There is also no requirement for the new programming language produced by this work to match existing programming languages. Ideally, the new system will be general and powerful enough that many people will achieve their objectives without having to move to other new languages. Hopefully, the need to learn some of the harder computer science concepts can be deferred or eliminated. For those who do move on to other languages or even to become computer scientists, their early success with this first language should ease their difficulties in learning the harder computer science concepts.

1.4 Thesis Statement

The thesis statement for this work is:

this user-centered design process, incorporating principles from human-computer interaction, psychology of programming, and empirical studies, will result in a unique programming system that is easier to learn and use than more conventional programming systems.

1.5 Target Audience and Domain

The target audience for my new programming system is children in fifth grade (about ten years old) or older. I chose to build a system for children because they often have an interest in learning how to program, but can be quickly discouraged when they try. Their goals are creative and ambitious – they would like to make programs that are similar to the applications they use, such as games and simulations. These applications are graphically rich and highly interactive, unlike the first programs they are likely to create in many professional programming systems, such as to display "hello world" on the screen. My goal is to provide an easy entry into creating these interactive graphical programs. However, to the extent possible, I also tried to create a general purpose language that scales well, so that it is not inherently limited to creating toy programs.

1.6 Understanding the Target Audience

In addition to general design principles that are applicable to all users, there is a wealth of information available about how beginner programmers work and the problems they encounter. This section summarizes the prior work and briefly describes the new studies I conducted to examine additional questions.

1.6.1 General Design Principles

The field of *Human Computer Interaction* (HCI) has general principles and heuristics that can be applied to programming system design [Nielsen 1994]:

- simple and natural dialog – user interfaces should be simplified, and should match the user's task in as natural a way as possible, such that the mapping between computer concepts and user concepts becomes straightforward.
- speak the user's language – the terminology in user interfaces should be based on the user's language, instead of using system-oriented terms or attaching non-standard meanings to familiar words.

- minimize user memory load – the system should take over the burden of memory from the user.
- consistency – the same command or action should always have the same effect.
- feedback – the system should continuously inform the user about what it is doing and how it is interpreting the user's input.
- clearly marked exits – the system should offer the user an easy way out of as many situations as possible, including ways to undo.
- shortcuts – the system should make it possible for experienced users to perform frequently used operations quickly.
- good error messages – the system should report errors politely in clear language, avoid obscure codes, use precise rather than vague or general explanations, and include constructive help for solving the problem.
- prevent errors – where possible, the user interface should be structured to avoid error situations.
- help and documentation – the help system and documentation should provide a quick way for users to find task-specific information when they are having a problem.

Many of these principles are routinely violated by programming systems – several examples are presented in Chapter 2.

When designing and evaluating programming systems, it is also useful to consider the more specific evaluation criteria in the *Cognitive Dimensions of Notations* framework (Cognitive Dimensions, for short) [Blackwell 2000, Green 1996]:

- viscosity – the system should not resist change; it should not require many user actions to accomplish one small goal.
- visibility – the information needed by the programmer at any particular time should be visible or very easy to access.
- premature commitment – the system should not force the user to go about the job in a particular order, or make a decision before the needed information is available.
- hidden dependencies – important links between entities should be visible.

- role expressiveness – the purpose of an entity should be readily apparent.
- error proneness – the notation should protect against slips and errors.
- closeness of mapping – the system's operations should closely match the way users think about problem solutions.
- secondary notation – the system should allow the programmer to communicate additional information with comments, typography, layout, etc.
- progressive evaluation – the system should permit users to test partial programs.
- diffuseness – small goals should not require extraordinarily long solutions or large amounts of screen space.
- provisionality – the system should allow the user to sketch out uncertain parts of their solution.
- hard mental operations – none of the system's operations should require great mental effort to use.
- consistency – similar notations should mean similar things, and vice versa.
- abstraction management – the system should provide a way to define new facilities or terms that allow the user to express ideas more clearly or succinctly, but it should not force users to use this capability right from the start.

These factors are sometimes in conflict, so improving the system along one dimension can result in reduced performance on another. Tradeoffs are necessary, and in making these tradeoffs it is useful to consider cognitive models and observations from empirical studies.

1.6.2 Observations about Existing Programming Languages

The principles of *simple and natural dialog, speak the user's language* and *closeness of mapping* are reinforced by cognitive models that define programming as a process where the user translates a mental plan into one that is compatible with the computer [Hoc 1990b]. The language should minimize the difficulty of this translation by providing operators that match those in the plan, including any that may be specific to the topic or domain of the program. "The closer the programming world is to the problem world, the easier the problem-solving ought to be.... Conventional textual languages are a long way from that goal"

[Green 1996, p. 146]. Hix & Hartson describe the general usability guideline to *use cognitive directness* [Hix 1993, p. 38] to "minimize the mental transformations that a user must make. Even small cognitive transformations by a user take effort away from the intended task." If the language does not provide these high-level operators, programmers have to assemble lower-level primitives to achieve their goals. This synthesis is one of the greatest cognitive barriers to programming [Lewis 1987].

Programmers are often required to think about algorithms and data in ways that are very different than the ways they already think about them in other contexts. For example, a typical C program to compute the sum of a list of numbers includes three kinds of parentheses and three kinds of assignment operators in five lines of code:

```
sum = 0;
for (i=0; i<numItems; i++) {
      sum += items[i];
}
return sum;
```

In contrast, this can be done in a spreadsheet with a single line of code using the *sum* operator [Green 1996]. The mismatch between the way a programmer thinks about a solution and the way it must be expressed in the programming language makes it more difficult not only for beginners to learn how to program, but also for people to carry out their programming tasks even after they become more experienced. One of the most common bugs among professional programmers using C and C++ is the accidental use of "=" (assignment) instead of "==" (equality test). This mistake is easy to make and difficult to find, not only because of typographic similarity, but also because "=" operator does indeed mean equality in other contexts such as mathematics.

Soloway, Bonar & Erlich [Soloway 1989a] found that the looping control structures provided by modern languages do not match the natural strategies that most people bring to the programming task. Furthermore, when novices are stumped they try to transfer their knowledge of natural language to the programming task. This often results in errors because the programming language defines these constructs in an incompatible way [Bonar 1989]. For example, *then* is interpreted as *afterwards* instead of *in these conditions*.

1.6.3 Naturalness

There are two ways to improve closeness of mapping. One is to teach people to think more like computers; the other is to make the programming system's operations match how users think. The latter approach is preferred in this thesis. A primary goal of my programming system is to support the *natural* ways that non-programmers think about problem solutions, instead of making them learn new and often unnatural ways to accomplish their objectives. In this context, natural means *expected or accepted*. If people have a viable approach to solving problems, the ideal programming system would support that solution directly, without requiring the programmer to learn anything new or perform additional work in translating their ideas into program code.

By this definition, naturalness is not universal for all humans. People from different backgrounds and cultures, or from different points in history, are likely to bring different expectations and methods to the programming task. Therefore, a programming system that is designed to be natural for a particular target audience is unlikely to be universally optimal. This is why identifying the target audience is an intrinsic part of the design process, and why the process itself is important. It will have to be applied over and over again, in order to best support the particular characteristics of the people who will use each new programming system.

Striving for naturalness does not necessarily imply that the programming language should use natural language. Programming languages that have adopted natural-language-like syntaxes, such as Cobol [Sammet 1981] and HyperTalk [Goodman 1987], still have many usability problems. For example, HyperTalk often violates the principle of consistency [Thimbleby 1992]. There are also many ambiguities in natural language that are resolved by humans through shared context and cooperative conversation [Grice 1975].

Novices attempt to enter into a human-like discourse with the computer, but programming languages systematically violate human conversational maxims because the computer cannot infer from context or enter into a clarification dialog [Pea 1986]. The use of natural language may compound this problem by making it more difficult for the user to understand the limits of the computer's intelligence [Nardi 1993].

However, these arguments do not imply that the algorithms and data structures should not be close to the ways people think about the problem. In fact, leveraging users' natural-language-like knowledge in a more formalized syntax can be an effective strategy for designing end-user-programming languages [Bruckman 1999].

There are additional motivations for why a more natural programming language might be better. A programming language is a type of user interface, and user interfaces in general are recommended to be *natural* so they are easier to learn and use, and will result in fewer errors. Naturalness is closely related to the concept of directness which, as part of *direct manipulation*, is a key principle in making user interfaces easier to use. Hutchins, Hollan & Norman describe *directness* as the distance between one's goals and the actions required by the system to achieve those goals [Hutchins 1986]. Reducing this distance makes systems more direct, and therefore easier to learn. User interface designers and researchers have been promoting directness at least since Shneiderman identified the concept [Shneiderman 1983], but it has not been a consideration in most programming language designs.

1.6.4 Studies of Naturalness in Problem Solving

This thesis presents two studies examining the language and structure that children and adults naturally use before they have been exposed to programming (Chapter 3). In these studies, I gave programming tasks to non-programmers and they solved these problems by writing and sketching their answers on paper. The tasks covered a broad set of essential programming techniques and concepts, such as control structures, storage and manipulation of data, arithmetic, Boolean logic, searching and sorting, animation, interactions among objects, etc. In posing the problems, I was careful to minimize the risk that my materials would influence the answers, so I used pictures and very terse captions.

Some observations from these studies were:

- An event-based or rule-based structure was often used, where actions were taken in response to events. For example, "when pacman loses all his lives, it's game over."
- Aggregate operators (acting on a set of objects all at once) were used much more often than iterating through the set and acting on the objects individually. For example, "Move everyone below the 5th place down by one."

- Participants did not construct complex data structures and traverse them, but instead performed content-based queries to obtain the necessary data when needed. For example, instead of maintaining a list of monsters and iterating through the list checking the color of each item, they would say "all of the blue monsters."
- A natural language style was used for arithmetic expressions. For example, "add 100 to score."
- Objects were expected to automatically remember their state (such as motion), and the participants only mentioned changes in this state. For example, "if pacman hits a wall, he stops."
- Operations were more consistent with list data structures, rather than arrays. For example, the participants did not create space before inserting a new object into the middle of a list.
- Participants rarely used Boolean expressions, but when they did they were likely to make errors. That is, their expressions were not correct if interpreted according to the rules of Boolean logic in most programming languages.
- Participants often drew pictures to sketch out the layout of the program, but resorted to text to describe actions and behaviors.

1.6.5 Study of Methods to Specify Queries

Because content-based queries were prevalent in non-programmers' problem solutions, I began to explore how this might be supported in a programming language. Queries are usually specified with Boolean expressions, and the accurate formulation of Boolean expressions has been a notorious problem in programming languages, as well as other areas such as database query tools [Hildreth 1988, Hoc 1989]. In reviewing prior research I found that there are few prescriptions for how to solve this problem effectively. For example, prior work suggests avoiding the use of the Boolean keywords *AND*, *OR*, and *NOT* [Greene 1990, McQuire 1995, Michard 1982], but does not recommend a suitable replacement query language.

Therefore I conducted a new study to examine the ways untrained children and adults naturally express and interpret queries, and to test a new tabular query form that I designed

called *match forms* (shown in Figure 1-1). This study confirmed that relying on the Boolean keywords, as well as parentheses for grouping, would result in poor usability. Textual alternatives that avoided the Boolean keywords were not reliably better. However, the match forms were successful.

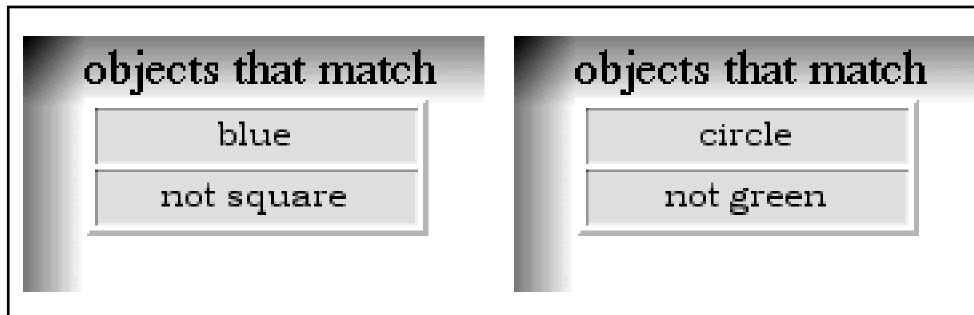

Figure 1-1. Match forms expressing the query: (blue and not square) or (circle and not green)

Each match form contains a vertical list of slots. Conjunction is specified by placing terms into these slots, one term per slot. Negation is performed by prefacing a term with the *NOT* operator, and disjunction is specified by placing additional match forms adjacent to the first one. This design avoids the need to name the *AND* and *OR* operators, provides a clear distinction between conjunction and disjunction, and makes grouping explicit. Match forms are suitable for incorporation into programming systems. When compared with textual Boolean expressions, users performed significantly better when they expressed their queries using match forms. When interpreting already-written queries, performance was about equal using either language. Chapter 4 contains full details about match forms and this study, as well as an application of this work to the search interface for the online *HCI Bibliography*.

1.6.6 Model of Computation

One of the biggest challenges for new programmers is to gain an accurate understanding of how computation takes place. Traditionally, programming is described to beginners in completely unfamiliar terms, often based on the von Neumann model, which has no real-world counterpart [du Boulay 1989a, du Boulay 1989b]. Beginners must learn, for example, that the program follows special rules of control flow for procedure calls and returns. There are complex rules that govern the lifetimes of variables and their scopes. Variables

may not exist at all when the program is not running, and during execution they are usually invisible, forcing the programmer to use print statements or debuggers to inspect them. This violates the principle of visibility, and contributes to a general problem of memory overload [Anderson 1985, Davies 1993].

Usability could be enhanced by providing a different model of computation that uses concrete and familiar terms [Mayer 1989, Smith 1994]. Using a different model of computation can have broad implications beyond beginners, because the model influences, and perhaps limits, how experienced programmers think about and describe computation [Stein 1999].

Section 1.7.1 introduces the new model of computation I invented to address this problem.

1.6.7 Visual vs. Textual

In visual languages, graphics replace some or all of the text in specifying programs. Proponents of visual programming languages often argue that reducing or eliminating the text in programming will improve usability [Smith 1994]. However, much of the underlying rationale for this expectation is suspect [Blackwell 1996]. User studies have shown mixed results on the superiority of visual languages over text (e.g. [Green 1992]), and the advantage of visual languages tends to diminish on larger tasks. It is useful to note that one of the most successful end-user programming systems to date is the spreadsheet, which is mostly textual [Nardi 1993].

My new programming system supports the hybrid graphical-textual approach used by the participants in my studies, and relies on the programming environment to alleviate some of the difficulties of textual languages. For example, during program entry, context-sensitive menus like those in Microsoft's Visual Studio can make it easier to know what choices are available and to help the user to enter the program correctly. This support could be augmented with a drag-and-drop syntax-directed editor, as seen in Squeak's eToys interface [Steinmetz 2001] and other systems. The system can also provide visual representations for textual elements that are difficult, such as the match forms mentioned in Section 1.6.5.

1.7 The HANDS Programming System Design

All of these observations have influenced the design of my new programming system, which is called HANDS (Human-centered Advances for the Novice Development of Soft-

ware). HANDS uses an event-based language that features a new concrete model for computation, provides queries and aggregate operators that match the way non-programmers express problem solutions, has high-visibility of program data, and includes domain-specific features for the creation of interactive animations and simulations. The HANDS system is detailed in Chapter 5.

1.7.1 Computational Model

In HANDS, the computation is represented as an agent named Handy, sitting at a table manipulating a set of cards (see Figure 1-2). All of the data in the system is stored on these cards, which are global, persistent and visible on the table. Each card has a unique name, and an unlimited set of name-value pairs, called properties. The program itself is stored in Handy's *thought bubble*. To emphasize the limited intelligence of the system, Handy is portrayed as an animal – like a dog that knows a few commands – instead of a person or a robot that could be interpreted as being very intelligent.

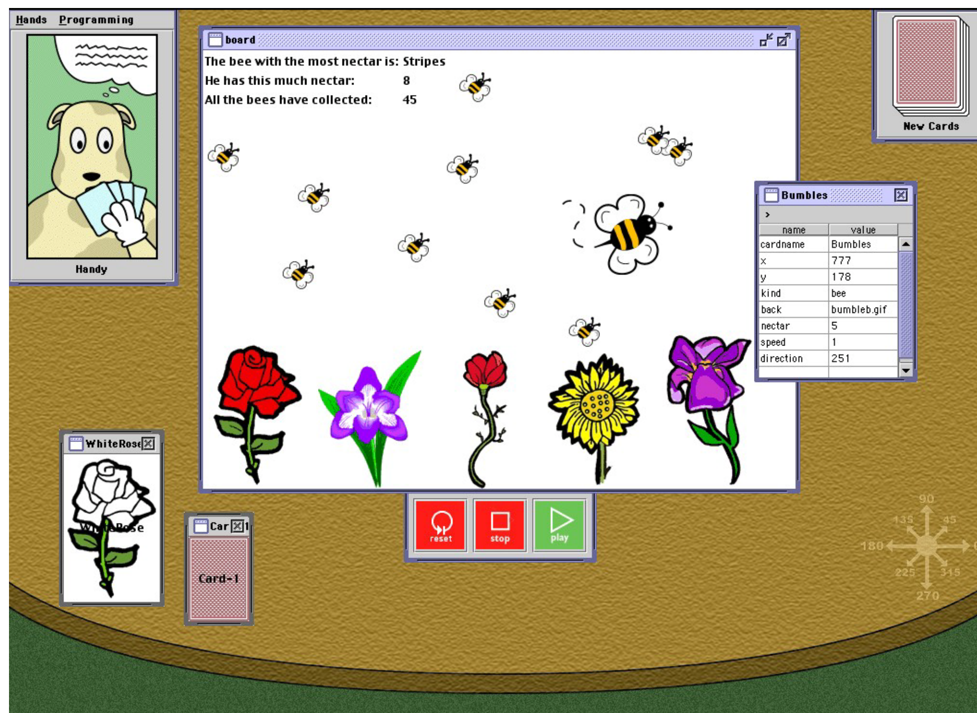

Figure 1-2. The HANDS system portrays the components of a program on a round table. All data is stored on cards, and the programmer inserts code into Handy's thought bubble at the upper left corner. When the play button is pressed, Handy begins responding to events by manipulating cards according to the instructions in the thought bubble. This is described in more detail in Chapter 5.

1.7.2 Programming Style and Model of Execution

HANDS is event-based, the programming style that most closely matches the problem solutions in my studies. A program is a collection of event handlers that are automatically called by the system when a matching event occurs. Inside an event handler, the programmer inserts the code Handy should execute in response to the event.

1.7.3 Aggregate Operations

In my studies, I observed that the participants used aggregate operators, manipulating whole sets of objects in one statement rather than iterating and acting on them individually. Many languages force users to perform iteration in situations where aggregate operations could accomplish the task more easily [Miller 1981]. Requiring users to translate a high-level aggregate operation into a lower-level iterative process violates the principle of *closeness of mapping*.

HANDS has full support for aggregate operations. All operators can accept lists as well as singletons as operands, or even one of each. For example,

- `1 + 1` evaluates to `2`
- `1 + (1,2,3)` evaluates to `2,3,4`
- `(1,2,3) + 1` evaluates to `2,3,4`
- `(1,2,3) + (2,3,4)` evaluates to `3,5,7`

1.7.4 Queries

In my studies, I observed that users do not maintain and traverse data structures. Instead, they perform queries to assemble lists of objects on demand. For example, they say “all of the blue monsters.” HANDS provides a query mechanism to support this. The query mechanism searches all of the cards for the ones matching the programmer’s criteria.

Queries begin with the word *all*. If a query contains a single value, it returns all of the cards that have that value in any property. Figure 1-3 contains cards representing three flowers and a bee to help illustrate the following queries.

- `all flowers` evaluates to `orchid, rose, tulip`

rose

name	value
cardname	rose
x	208
y	80
group	flower
nectar	100

tulip

name	value
cardname	tulip
x	350
y	80
group	flower
nectar	150

orchid

name	value
cardname	orchid
x	490
y	80
group	flower
nectar	75

bumble

name	value
cardname	bumble
x	636
y	80
group	bee
nectar	0

Figure 1-3. When the system evaluates the query `all flowers` it returns `rose, tulip, orchid`

- `all bees` evaluates to `bumble`
- `all snakes` evaluates to the empty list

HANDS permits more complex queries to be specified with traditional Boolean expressions, however the intention is to eventually incorporate match forms into the system as an option for specifying and displaying queries.

Queries and aggregate operations work in tandem to permit the programmer to concisely express actions that would require iteration in most languages. For example,

- `set the nectar of all flowers to 0`

1.7.5 Domain-Specific Support

HANDS has domain-specific features that enable programmers to easily create highly-interactive graphical programs. For example, the system's suite of events directly supports this class of programs. The system automatically detects collisions among objects and generates events to report them to the programmer. It also generates events in response to input from the user via the keyboard and mouse. It is easy to create graphical objects and text on the screen, and animation can be accomplished without any programming.

1.8 Evaluation

To examine the effectiveness of three key features of HANDS: *queries, aggregate operations, and data visibility*, I conducted a study comparing the system with a limited version that lacks these features. In the limited version, programmers could achieve the same results but had to use more traditional programming techniques. Fifth-grade children were

able to learn the HANDS system during a three-hour session, and then use it to solve programming problems. Children using the full-featured HANDS system performed significantly better than their peers who used the reduced-feature version. This is evidence that this set of features improves usability over the typical set of features in programming systems.

In a separate informal study, a high-school student compared hands with Stagecast, a commercial programming environment for children [Earhart 1999]. He implemented a game in both systems, and concluded that HANDS was easier to use, enabled him to implement more features, and required fewer lines of code. In addition, several more experienced programmers have used HANDS to implement a broad variety of programs to explore its range of capabilities.

Evaluation of the HANDS system is detailed in Chapter 6.

1.9 Contributions

The contributions of this thesis are:

- a case study of a new *design process* for creating programming systems, where usability is a first class objective;
- the *HANDS programming system* for children, which has a unique set of features due to its user-centered design, several of which were demonstrated to be more usable than those found in typical programming systems;
- a new *model of computation*, or way of thinking about programs, that is concrete and based on familiar concepts, unlike the traditional Turing machine or von Neumann machine models;
- a general-purpose *programming language* that offers database-style access to the program's data, and in which all operators can be applied to singletons and lists;
- *match forms*, a tabular method for expressing queries that was compared to textual expressions and shown to improve beginners' performance;
- a new *query interface* for the HCI Bibliography (www.hcibib.org), based on match forms, which reduces user errors in comparison to the old interface;

- *empirical evidence* about how non-programmers express problem solutions, which can be used to help designers generate and select programming system features that provide a close mapping between those problem solutions and their expression in program code;
- *empirical evidence* characterizing the kinds of errors made by inexperienced users of textual Boolean expressions;
- *a user study* demonstrating the effectiveness of queries, aggregate operations, and high-visibility of data, in comparison to the typical features sets of programming systems; and,
- a *broad survey* of the prior work on beginner programmers, organized in a form that can be used by other programming system designers (appears in Appendix C).

1.10 Overview of Thesis

The remainder of this thesis is organized as follows: Chapter 2 describes the prior empirical work on beginner programmers as well as other programming systems for beginners and children; Chapter 3 describes the first two studies examining the language and structure in non-programmers solutions to programming problems; Chapter 4 describes the third study, examining methods for specifying queries, and provides details about match forms; Chapter 5 details the design of the HANDS system; Chapter 6 describes a fourth study, to evaluate features of HANDS, as well as other less formal evaluations; Chapter 7 discusses the implications of this work and some ideas for future work; and Chapter 8 gives some concluding remarks.

Supplemental materials are contained in appendices. Appendix A contains a formal specification of the HANDS language syntax; Appendix B contains some example programs implemented in HANDS; Appendix C contains the full text of my technical report surveying usability issues in programming systems for beginners; Appendix D contains the materials used in the first study; Appendix E contains the materials used in the second study; Appendix F contains the materials used in the third study; and Appendix G contains the materials used in the fourth study.

This document contains only Chapter 1 of the thesis.
For citation, please refer to the full thesis document, which is available at:
http://www.cs.cmu.edu/~pane/thesis

CHAPTER 9 References

Anderson, J.R. and 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. and 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. Brussels, Belgium: 135-150.

Baraff, D. (1989). "Analytical Methods for Dynamic Simulation of Non-Penetrating Rigid Bodies." Computer Graphics 23(3): 223-232.

Biermann, A.W., Ballard, B.W. and Sigmon, A.H. (1983). "An Experimental Study of Natural Language Programming." International Journal of Man-Machine Studies 18(1): 71-87.

Blackwell, A.F. (1996). Metacognitive Theories of Visual Programming: What Do We Think We Are Doing? Proceedings of the VL'96 IEEE Symposium on Visual Languages. Boulder, CO, IEEE Computer Society Press: 240-246.

Blackwell, A.F. and Green, T.R.G. (2000). A Cognitive Dimensions Questionnaire Optimised for Users. Proceedings of the 12th Annual Meeting of the Psychology of Pro-

grammers Interest Group. A. F. Blackwell and E. Bilotta. Corigliano Calabro, Italy, Edizioni Memoria: 137-154.

Bonar, J. and Soloway, E. (1989). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 325-353.

Bonar, J.G. and Cunningham, R. (1988). Bridge: Tutoring the Programming Process. Intelligent Tutoring Systems: Lessons Learned. J. Psotka, L. D. Massey and S. A. Mutter. Hillsdale, NJ, Lawrence Erlbaum Associates: 409-434.

Bourne, L.E. (1966). Human Conceptual Behavior. Boston, Allyn & Bacon.

Bruckman, A. and Edwards, E. (1999). Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. Proceedings of the 1999 Conference on Human Factors in Computing Systems. Pittsburgh, PA, ACM Press: 207-214.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. (1997). "Mini-languages: A Way to Learn Programming Principles." Education and Information Technologies 2(1): 65-83.

Carriero, N. and Gelernter, D. (1989). "Linda in Context." Communications of the ACM 32(4): 444-458.

Carver, N. and Lesser, V. (1994). "The Evolution of Blackboard Control Architectures." Expert Systems with Applications 7(1): 1-30.

Conway, D.M. (1998). An Algorithmic Approach to English Pluralization. Proceedings of the Second Annual Perl Conference. C. Salzenberg. San Jose, CA, O'Reilly.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., Durbin, J., Gossweiler, R., Koga, S., Long, C., Mallory, B., Miale, S., Monkaitis, K., Patten, J., Pierce, J., Shochet, J., Staack, D., Stearns, B., Stoakley, R., Sturgill, C., Viega, J., White, J., Williams, G. and Pausch, R. (2000). Alice: Lessons Learned from Building a 3D System for Novices. Proceedings of CHI2000 Conference on Human Factors in Computing Systems. T. Turner and G. Szwillis. The Hague, Netherlands, ACM Press: 486-493.

Conway, M.J. (1997). Alice: Easy-to-Learn 3D Scripting for Novices. Ph.D. Thesis. University of Virginia. School of Engineering and Applied Science, 242 pages.

Cordy, J.R. (1992). Hints on the Design of User Interface Language Features – Lessons from the Design of Turing. Languages for Developing User Interfaces. B. A. Myers. Boston, Jones and Bartlett: 329-340.

Cypher, A. and Smith, D.C. (1995). KidSim: End User Programming of Simulations. Proceedings of CHI'95 Conference on Human Factors in Computing Systems. Denver, ACM: 27-34.

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

Détienne, F. (1990). Difficulties in Designing with an Object-Oriented Programming Language: An Empirical Study. Proceedings of INTERACT '90 Conference on Computer-Human Factors. Cambridge, England: 971-976.

Détienne, F. (2001). Software Design: Cognitive Aspects. London, Springer.

DiGiano, C., Kahn, K., Cypher, A. and Smith, D.C. (2001). "Integrating Learning Supports into the Design of Visual Programming Systems." Journal of Visual Languages & Computing 12(5): 501-524.

DiGiano, C.J. (1996). Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming. Boulder, CO, University of Colorado**:** Department of Computer Science Technical Report CU-CS-822-96, 154 pages.

diSessa, A.A. and Abelson, H. (1986). "Boxer: A Reconstructible Computational Medium." Communications of the ACM 29(9): 859-868.

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

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

Earhart, C., Ed. (1999). Stagecast Creator Teacher's Guide. Palo Alto, CA, Stagecast Software, http://www.stagecast.com.

Essens, P.J.M.D., McCann, C.A. and Hartevelt, M.A. (1992). An Experimental Study of the Interpretation of Logical Operators in Database Querying. Cognitive Ergonomics: Contributions from Experimental Psychology. G. C. v. d. Veer, S. Bagnara and G. A. M. Kempen. Amsterdam, North-Holland, Elsevier Science Publishers: 201-225.

Finzer, W.F. and Gould, L. (1993). Rehearsal World: Programming by Rehearsal. Watch What I Do: Programming by Demonstration. A. Cypher, MIT Press.

Galotti, K.M. and Ganong, W.F., III (1985). “What Non-Programmers Know About Programming: Natural Language Procedure Specification.” International Journal of Man-Machine Studies 22: 1-10.

Glass, R.L. (1995). “OO Claims – Naturalness, Seamlessness Seem Doubtful.” Software Practitioner 5(2).

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

Gould, L. and Finzer, W. (1984). “Programming by Rehearsal.” BYTE Magazine 9(6).

Green, T.R.G. (1990). The Nature of Programming. Psychology of Programming. J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. London, Academic Press: 21-44.

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

Green, T.R.G. and 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. and Gomez, L.M. (1990). “No IFs, ANDs, or ORs: A Study of Database Querying.” International Journal of Man-Machine Studies 32(3): 303-326.

Grice, H.P. (1975). Logic and Conversation. Syntax and Semantics III: Speech Acts. P. Cole and J. Morgan. New York, Academic Press.

Gross, P. (1999). Director 7 and Lingo Authorized, Peachpit Press.

Hays, J.G. and Burnett, M.M. (1995). A Guided Tour of Forms/3, Oregon State University: Dept. of Computer Science Technical Report 95-60-6.

Hildreth, C. (1988). Intelligent Interfaces and Retrieval methods for Subject Search in Bibliographic Retrieval Systems. Research, Education, Analysis & Design. Springfield, IL.

Hix, D. and Hartson, H.R. (1993). Developing User Interfaces: Ensuring Usability Through Product and Process. New York, New York, John Wiley & Sons, Inc.

Hoc, J.-M. (1989). Do We Really Have Conditional Statements in Our Brains? Studying the Novice Programmer. E. Soloway and J. C. Spohrer. Hillsdale, NJ, Lawrence Erlbaum Associates: 179-90.

Hoc, J.-M., Green, T.R.G., Samurçay, R. and Gilmore, D.J., Eds. (1990a). Psychology of Programming. Computers and People Series. London, Academic Press.

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

Hutchins, E.L., Hollan, J.D. and Norman, D.A. (1986). Direct Manipulation Interfaces. Hillsdale, NJ, Lawrence Erlbaum Associates.

Ingalls, D.H.H. (1981). Design Principles Behind Smalltalk. BYTE Magazine, August 1981.

Joers, J. (1999). Stagecast Creator Creator's Guide. Palo Alto, CA, Stagecast Software, http://www.stagecast.com/.

Jones, S. (1998). Graphical Query Specification and Dynamic Result Previews for a Digital Library. Proceedings of the ACM Symposium on User Interface Software and Technology: 143-151.

Kahn, K. (1996). “ToonTalk: An Animated Programming Environment for Children.” Journal of Visual Languages and Computing 7(2): 197-217.

Kahn, K. (1999). From Prolog and Zelda to ToonTalk. Proceedings of the 1999 International Conference on Logic Programming. D. De Schreye, MIT Press.

Kohl, A. and Rupietta, W. (1987). The Natural Language Metaphor: An Approach to Avoid Misleading Expectations. Proceedings of IFIP INTERACT'87: Human-Computer Interaction: 555-560.

Lewis, C. and Olson, G.M. (1987). Can Principles of Cognition Lower the Barriers to Programming? Empirical Studies of Programmers: Second Workshop. G. M. Olson, S. Sheppard and E. Soloway. Norwood, NJ, Ablex: 248-263.

Martin, F.G. and Resnick, M. (1993). LEGO/Logo and Electronic Bricks: Creating a Scienceland for Children. Advanced Educational Technologies for Mathematics and Science. D. L. Ferguson. Berlin, Springer-Verlag.

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

McDaniel, R. (1999). Building Whole Applications Using Only Programming-by-Demonstration. Ph.D. Thesis. Carnegie Mellon University. Computer Science Department. Pittsburgh, PA, 271 pages.

McIver, L.K. (2001). Syntactic and Semantic Issues in Introductory Programming Education. Ph.D. Thesis. Monash University. School of Computer Science and Software Engineering. Australia, 200 pages.

McQuire, A. and 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: 373.

Michard, A. (1982). “Graphical Presentation of Boolean Expressions in a Database Query Language: Design Notes and an Ergonomic Evaluation.” Behaviour and Information Technology 1(3): 279-288.

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.

Miller, P., Pane, J., Meter, G. and Vorthmann, S. (1994). "Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University." Interactive Learning Environments 4(2): 140-158.

Modugno, F. (1995). Extending End-User Programming in a Visual Shell with Programming by Demonstration and Graphical Language Techniques. Ph.D. Thesis. Carnegie Mellon University. Computer Science Department. Pittsburgh, PA, 334 pages.

Modugno, F., Corbett, A.T. and Myers, B.A. (1996). Evaluating Program Representation in a Visual Shell. Empirical Studies of Programmers: Sixth Workshop. W. D. Gray and D. A. Boehm-Davis. Norwood, NJ, Ablex Publishing Corporation: 131-146.

Mulholland, P. and Watt, S.N.K. (2000). "Learning by Building: A Visual Modelling Language for Psychology Students." Journal of Visual Languages and Computing 11(5): 481-504.

Myers, B.A. (1992). "Demonstrational Interfaces: A Step Beyond Direct Manipulation." IEEE Computer 25(8): 61-73.

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

Newell, A. and 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. Usability Inspection Methods. J. Nielsen and R. L. Mack. New York, John Wiley & Sons: 25-62.

Pane, J.F. and Myers, B.A. (1996). Usability Issues in the Design of Novice Programming Systems. Pittsburgh, PA, Carnegie Mellon University**:** School of Computer Science Technical Report CMU-CS-96-132, 85 pages.

Pane, J.F. and Myers, B.A. (2000). Tabular and Textual Methods for Selecting Objects from a Group. Proceedings of VL 2000: IEEE International Symposium on Visual Languages. Seattle, WA, IEEE Computer Society: 157-164.

Pane, J.F., Ratanamahatana, C.A. and Myers, B.A. (2001). "Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems." International Journal of Human-Computer Studies 54(2): 237-264.

Papert, S. (1980). Mindstorms: Children, Computers, and Powerful Ideas. New York, Basic Books.

Pattis, R.E., Roberts, J. and Stehlik, M. (1995). Karel the Robot: A Gentle Introduction to the Art of Programming. New York, John Wiley & Sons.

Pea, R. (1986). "Language-Independent Conceptual "Bugs" in Novice Programming." Journal of Educational Computing Research 2(1).

Pictorius (1996). Prograph CPX User Guide. Halifax, Nova Scotia, Pictorius Incorporated, http://www.pictorius.com/prograph.html.

Repenning, A. (2000). AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents. Interaction 2000, Tokyo, Japan.

Repenning, A. and Sumner, T. (1995). "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages." Computer 28: 17-25.

Resnick, M. (1994). Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds. Boston, The MIT Press.

Sammet, J.E. (1981). The Early History of COBOL. History of Programming Languages. R. Wexelblat. New York, Academic Press.

Sherwood, B.A. (1988). The cT Language. Champaigne, IL, Stipes Publishing Company.

Shneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages." IEEE Computer 16(8): 57-69.

Smith, D.C. (2000). "Building Personal Tools by Programming." Communications of the ACM 43(8): 92-95.

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

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

Soloway, E. and Spohrer, J.C., Eds. (1989b). Studying the Novice Programmer. Hillsdale, NJ, Lawrence Erlbaum Associates.

Spohrer, J.G. and Soloway, E. (1986). Analyzing the High Frequency Bugs in Novice Programs. Empirical Studies of Programmers. E. Soloway and S. Iyengar. Washington, DC, Ablex Publishing Corporation: 230-251.

Stein, L.A. (1999). "Challenging the Computational Metaphor: Implications for How We Think." Cybernetics and Systems 30(6): 473-507.

Steinmetz, J. (2001). Computers and Squeak as Environments for Learning. Squeak: Open Personal Computing and Multimedia. M. Guzdial and K. Rose, Prentice Hall: 453-482.

Tanaka, J. (1999). The Perfect Search. Newsweek. 134: 71, September 27 1999.

Teitelman, W. and Masinter, L. (1981). "The Interlisp Programming Environment." Computer 14(4): 25-34.

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

Thomas, J. and Gould, J. (1975). A Psychological Study of Query by Example. National Computer Conference. Anaheim, CA, AFIPS. 44: 439-445.

Turtle, H. (1994). Natural Language vs. Boolean Query Evaluation: A Comparison of Retrieval Performance. Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval: 212-220.

Wason, P.C. (1959). "The Processing of Positive and Negative Information." Quarterly Journal of Experimental Psychology 11.

Webgain (2001). JavaCC - The Java Parser Generator, http://www.webgain.com/products/metamata/java_doc.html.

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

Wilcox, E. and Burnett, M. Programming a Single Digit LED in Forms/3 http://www.cs.orst.edu/~burnett/Forms3/LED.html.

Young, D. and 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.

This document contains only Chapter 1 of the thesis.
For citation, please refer to the full thesis document, which is available at:
http://www.cs.cmu.edu/~pane/thesis