Chapter 3

More Natural Programming Languages and Environments

JOHN F. PANE[1] and BRAD A. MYERS[2]
[1]*RAND Corporation, jpane@rand.org*
[2]*Carnegie Mellon University, bam+@cs.cmu.edu*

Abstract. Over the last six years, we have been working to create programming languages and environments that are more *natural*, by which we mean closer to the way people think about their tasks. The goal is to make it possible for people to express their ideas in the same way they think about them. To achieve this, we performed various studies about how people think about programming tasks, and then used this knowledge to develop a new programming language and environment called HANDS. This chapter provides an overview of the goals and background for the Natural Programming research, the results of some of our user studies, and the highlights of the language design.

1. Introduction

The Natural Programming Project is studying ways to make learning to program significantly easier, so that more people will be able to create useful, interesting, and sophisticated programs. The goals of this project are to define and use a new programming language design process; where we study how non-programmers reason about programming concepts, create new programming languages and environments that take advantage of these findings, and evaluate them. It is somewhat surprising that in spite of over 30 years of research in the areas of empirical studies of programmers (ESP) and human–computer interaction (HCI), the designs of new programming languages have generally not taken advantage of what has been discovered. For example, the new C#, Java, and JavaScript languages use the same mechanisms for looping, conditionals, and assignments that have been shown to cause many errors for both beginning and expert programmers in the C language. Our thorough investigation of the ESP and HCI literature has revealed many results which can be used to guide the design of a new programming system, many of which have not been utilized in previous designs. However, there are many significant "holes" in the knowledge about how people reason about programs and programming. For example, research about which fundamental paradigms of computing are the most natural has not been conclusive. We are performing user studies to investigate this question. Another issue is that most of the prior research has studied people using existing languages, and so there is little information about how people might express various concepts if not restricted by these language designs.

In the context of this prior work, as well as best practices in user-centered design, we adopted a *Natural Programming* design process, which treats usability as a first-class objective in programming system design by following these steps:

- *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.
- *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.
- *Design the new system* based on this information.
- *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, the prior knowledge about the human aspects of programming is considered, and the strategy for addressing any unanswered questions is to conduct user studies to obtain design guidance and to assess prototypes.

This chapter summarizes the results to date for the Natural Programming project. More details were reported by Pane (2002), as well as in many other papers that are available from our web site: (http://www.cs.cmu.edu/~NatProg). First, we discuss why naturalness might be better for developers, and then discuss a survey of prior work as it relates to the design of more natural programming languages and environments. Then we discuss three studies we performed to evaluate what might be more natural in programs for graphics and data processing. The results were used in the design of a new language and environment called HANDS (Human-centered Advances for the Novice Development of Software). A user study showed that the novel aspects of HANDS were helpful to novice fifth graders writing their first programs. Finally, we discuss the current work of the Natural Programming project on making the debugging process more natural.

2. Why Natural Might be Better for End-User Developers

In Natural Programming we aim for the programming system to work in the way that people expect, especially end-user developers who may have little or no formal training or experience in programming. The premise of this approach is that the developers will have an easier job if their programming tasks are made more natural. We define *natural* as "faithfully representing nature or life."

Why would this make end-user programming easier? One way to define programming is the process of transforming a mental plan in familiar terms into one that is compatible with the computer (Hoc and Nguyen-Xuan, 1990). The closer the language is to the programmer's original plan, the easier this refinement process will be. This is closely related to the concept of *directness* that, as part of "direct manipulation," is a key principle in making user interfaces easier to use. Hutchins et al. (1986) describe directness as the distance between one's goals and the actions required by the system to achieve those goals. 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 (1983) identified the concept, but it has not even been a consideration in most programming language designs. Green and Petre (1996, p. 146) also argue in favor of directness, which they call *closeness of mapping:* "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."

User interfaces in general are also recommended to be natural so they are easier to learn and use, and will result in fewer errors. For example, Nielsen (1993, p. 126) recommends that user interfaces should "speak the user's language," which includes having good mappings between the user's conceptual model of the information and the computer's interface for it. One of Hix and Hartson's usability guidelines is to "use cognitive directness," which means to "minimize the mental transformations that a user must make. Even small cognitive transformations by a user take effort away from the intended task." Conventional programming languages do not provide the high-level operators that would provide this directness, instead requiring the programmer to make transformations from the intended task to a code design that assembles lower-level primitives.

Norman also discusses the conceptual gap between the representations that people use in their minds and the representations that are required to enter these into a computer. He calls these the "gulfs of execution and evaluation." He says; "there are only two ways to . . . bridge the gap between goals and system: move the system closer to the user; [or] move the user closer to the system" (Norman, 1986, p. 43). We argue that if the computer language expressed algorithms and data in a way that was closer to people's natural expressions, the gaps would be smaller. As Smith et al. (1996, p. 60) have said, "regardless of the approach, with respect to programming, trying to move most people closer to the system has not worked."

The proposed research is closely aligned with the concept of "Gentle Slope Systems" (MacLean et al., 1990; Myers et al., 1992), which are systems where for each incremental increase in the level of customizability, the user only needs to learn an incremental amount. This is contrasted with most systems, which have "walls" where the user must stop and learn many new concepts and techniques to make further progress (see Figure 3.1). We believe that systems can use direct manipulation and demonstrational techniques, where users give examples of the desired outcome (Myers, 1992), to lower the initial starting point and enable users to get useful work done immediately. Systems and languages can also be self-disclosing and easy to learn, so the number and height of the walls is minimized, if they cannot be eliminated entirely.

We note that a programming system that is designed to be natural for a particular target audience may not be universally optimal. People of different ages, from different backgrounds and cultures, or from different points in history, are likely to bring different expectations and methods to the programming task. This is why identifying the target

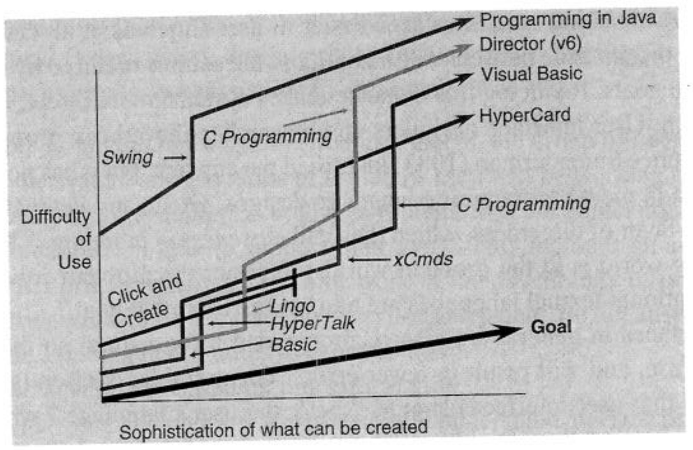


Figure 3.1. The ideal of a gentle slope system. The intent of this graph is to portray how difficult it is to use various tools to create customizations of different levels of sophistication. For example, with Java, it is quite hard to get started, so the Y intercept is high up. The vertical walls are where the designer needs to stop and learn something entirely new. For Java, the wall is where the user needs to learn Swing to do graphics. With Visual Basic, it is easier to get started, so the Y intercept is lower, but Visual Basic has two walls—one when you have to learn the Basic programming language, and another when you have to learn C programming because Visual Basic is no longer adequate. Click and Create was a menu based tool from Corel, and its line stops because it does not have an extension language, and you can only do what is available from the menus and dialog boxes.

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.

3. Survey of Earlier Work

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 and Petre, 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 the "=" operator does indeed mean equality in other contexts such as mathematics.

Soloway et al. (1989) 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 and Soloway, 1989). For example, "then" is interpreted as meaning "afterwards" instead of "in these conditions."

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, 1989; du Boulay et al., 1989). 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 and Jeffries, 1985; Davies, 1993).

Usability could be enhanced by providing a different model of computation that uses concrete and familiar terms (Mayer, 1989; Smith et al., 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).

In the 1970s, Miller (1974; 1981) examined natural language procedural instructions generated by non-programmers and made a rich set of observations about how the participants naturally expressed their solutions. This work resulted in a set of recommended features for computer languages. For example, Miller suggested that contextual referencing would be a useful alternative to the usual methods of locating data objects by using variables and traversing data structures. In contextual referencing, the programmer identifies data objects by using pronouns, ordinal position, salient or unique features, relative referencing, or collective referencing (Miller, 1981, p.213).

Although Miller's approach provided many insights into the natural tendencies of non-programmers, there have only been a few studies that have replicated or extended that work. Biermann et al. (1983) confirmed that there are many regularities in the way people express step-by-step natural language procedures, suggesting that these regularities could be exploited in programming languages. Galotti and Ganong (1985) found that they were able to improve the precision in users' natural language specifications by ensuring that the users understood the limited intelligence of the recipient of the instructions. Bonar and Cunningham (1988) found that when users translated their natural-language specifications into a programming language, they tended to use the natural-language semantics even when they were incorrect for the programming language. It is surprising that the findings from these studies have apparently had little impact on the designs of new programming languages that have been invented since then.

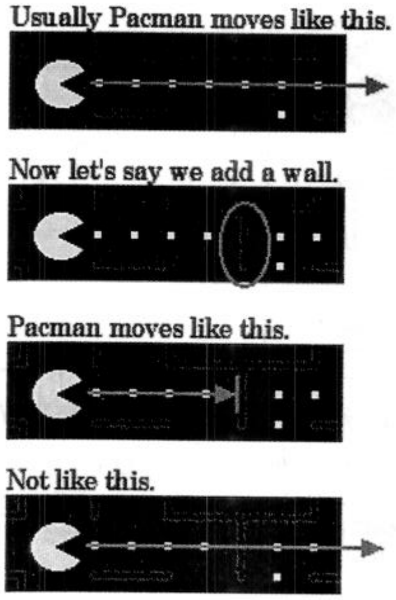
4. Initial User Studies

We conducted two studies that were loosely based on Miller's work, to examine the language and structure that children and adults naturally use before they have been exposed to programming. A risk in designing these studies is that the experimenter could bias the participants with the language used in asking the questions. For example, the experimenter cannot just ask: "How would you tell the monsters to turn blue when the PacMan eats a power pill?" because this may lead the participants to simply parrot parts of the question back in their answers. This would defeat the prime objective of these studies, which is to examine users' unbiased responses. Therefore our materials were constructed with great care to minimize this kind of bias, with terse descriptions and graphical depictions of the problem scenarios.

In our studies, participants were presented with programming tasks and asked to solve them on paper using whatever diagrams and text they wanted to use. Before designing the tasks, a list of essential programming techniques and concepts was enumerated, covering various kinds of applications. These include: use of variables, assignment of values, initialization, comparison of values, Boolean logic, incrementing and decrementing of counters, arithmetic, iteration and looping, conditionals and other flow control, searching and sorting, animation, multiple things happening simultaneously (parallelism), collisions and interactions among objects, and response to user input.

Because children often express interest in creating games and animated stories, the first study focused on the skills that are necessary to build such programs. In this study, the PacMan video game was chosen as a fertile source of interesting problems that require these skills. Instead of asking the participants to implement an entire PacMan game, various situations were selected from the game because they touch upon one or more of the above concepts. This allowed a relatively small set of exercises to broadly cover as many of the concepts as possible in the limited amount of time available. Many of the skills that were not covered in the first study were covered in a second study, which used a set of spreadsheet-like tasks involving database manipulation and numeric computation.

A set of nine scenarios from the PacMan game were chosen, and graphical depictions of these scenarios were developed, containing still images or animations and a minimal amount of text. The topics of the scenarios were: an overall summary of the game, how the user controls PacMan's actions, PacMan's behavior in the presence and absence of other objects such as walls, what should happen when PacMan encounters a monster under various conditions, what happens when PacMan eats a power pill,



Usually Pacman moves like this.

Now let's say we add a wall.

Pacman moves like this.

Not like this.

Do this: Write a statement that summarizes how I (as the computer) should move Pacman in relation to the presence or absence of other things.

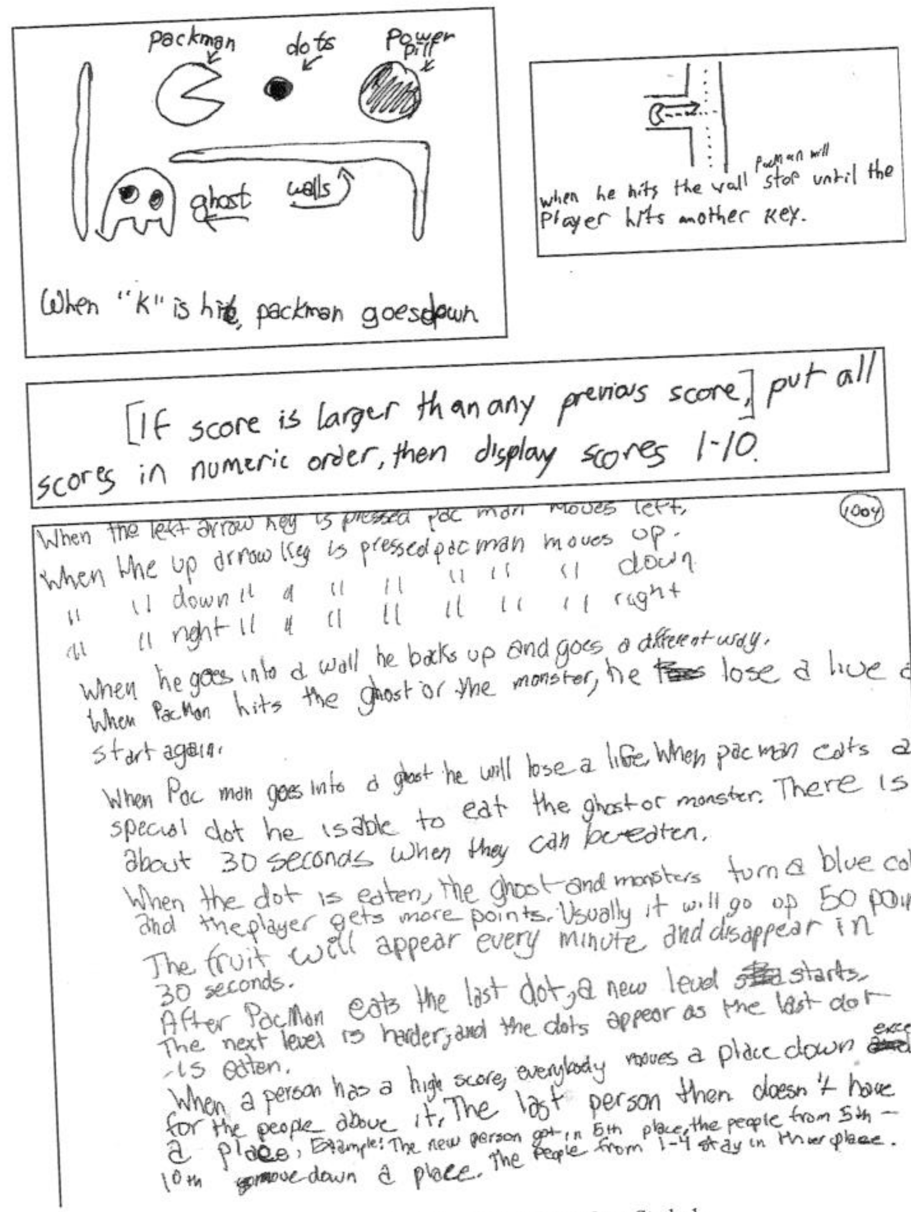
Figure 3.2. Depiction of a problem scenario in study one.

scorekeeping, the appearance and disappearance of fruit in the game, the completion of one level and the start of the next, and maintenance of the high score list. Figure 3.2 shows one of the scenario depictions. Figure 3.3 shows excerpts from participants' solutions.

We developed a rating form to be used by independent analysts to classify each participant's responses (Figure 3.4). Each question on the form addressed some facet of the participant's problem solution, such as the way a particular word or phrase was used, or some other characteristic of the language or strategy that was employed.

Each question was followed by several categories into which the participant's responses could be classified. The analyst was instructed to look for relevant sentences in the participant's solution, and classify each one by placing a tick mark in the appropriate category, also noting which problem the participant was answering when the sentence was generated. Each question also had an "other" category, which the rater marked when the participant's utterance did not fall into any of the supplied categories. When they did this, they added a brief comment.

To see whether the observations from the first study would generalize to other domains and other age groups, a second study was conducted. This study used database access scenarios that are more typical of business programming tasks, and was administered to a group of adults as well as a group of children.


packman
dots
Power pill
ghost
walls
Pacman will

When "K" is hit, packman goesdown

when he hits the wall stop until the Player hits another key.

[If score is larger than any previous score] put all scores in numeric order, then display scores 1-10.

(1004)

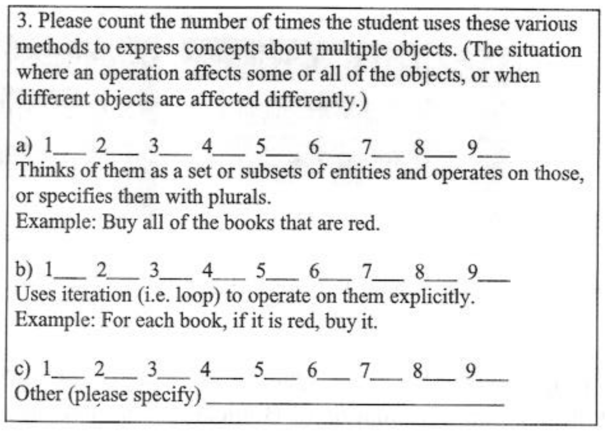
When the left arrow key is pressed pac man moves left.
When the up arrow key is pressed pacman moves up.
" " down " " " " " " " down
" " right " " " " " " " right

When he goes into a wall he backs up and goes a different way.
When PacMan hits the ghost or the monster, he ~~has~~ lose a live and start again.
When Pac man goes into a ghost he will lose a life. When pacman eats a special dot he is able to eat the ghost or monster. There is about 30 seconds when they can be eaten.
When the dot is eaten, the ghost and monsters turn a blue color and the player gets more points. Usually it will go up 50 points.
The fruit will appear every minute and disappear in 30 seconds.
After PacMan eats the last dot, a new level ~~st~~ starts. The next level is harder, and the dots appear as the last dot is eaten.
When a person has a high score, everybody moves a place down except ~~and~~ for the people above it. The last person then doesn't have a place. Example: The new person got in 5th place, the people from 5th – 10th ~~go~~move down a place. The people from 1-4 stay in their place.

Figure 3.3. Excerpts from participants' solutions to problems from Study 1.

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, its game over."



3. Please count the number of times the student uses these various methods to express concepts about multiple objects. (The situation where an operation affects some or all of the objects, or when different objects are affected differently.)

a) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Thinks of them as a set or subsets of entities and operates on those, or specifies them with plurals.
Example: Buy all of the books that are red.

b) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Uses iteration (i.e. loop) to operate on them explicitly.
Example: For each book, if it is red, buy it.

c) 1___ 2___ 3___ 4___ 5___ 6___ 7___ 8___ 9___
Other (please specify) __________________________

Figure 3.4. A question from the rating form for study one. The nine blanks on each line correspond to the nine tasks that the participants solved.

- Aggregate operations, where a set of objects is acted upon all at once, were used much more often than iteration through the set to act 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 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 used 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.

Additional details about these studies were reported by Pane et al. (2001).

5. Studying the Construction of Sets

Because operations on groups of objects and content-based queries were prevalent in non-programmers' problem solutions, we began to explore how this might be supported

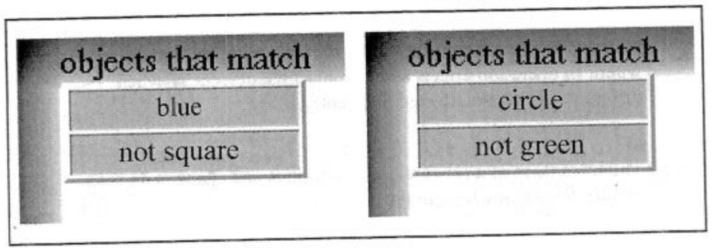


Figure 3.5. Match forms expressing the query: `(blue and not square) or (circle and not green)`.

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 we 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 et al., 1990; McQuire and Eastman, 1995; Michard, 1982), but does not recommend a suitable replacement query language.

Therefore we 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 we designed.

Although some graphical query methods had been shown to be more effective than Boolean expressions, many of them were limited to expressing very simple queries. We wanted a solution that is fully expressive. Also, many of the graphical systems would not integrate well into a programming language, where the entire computer screen cannot be devoted to this one subtask of the programming process. We required a format that is compact and readable in the context of a larger program. With these points in mind, we designed a tabular form that is fully expressive and compatible with the programming language we were developing.

Since we were planning to represent data on cards containing attribute-value pairs in the HANDS programming language, 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 3.5). 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 compact enough to be suitable for incorporation into programming systems.

The study used a grid of nine colored shapes, where a subset of the shapes could be marked (see Figure 3.6). Children and adults who participated in this study were given two kinds of problems: code generation problems, where some shapes were already

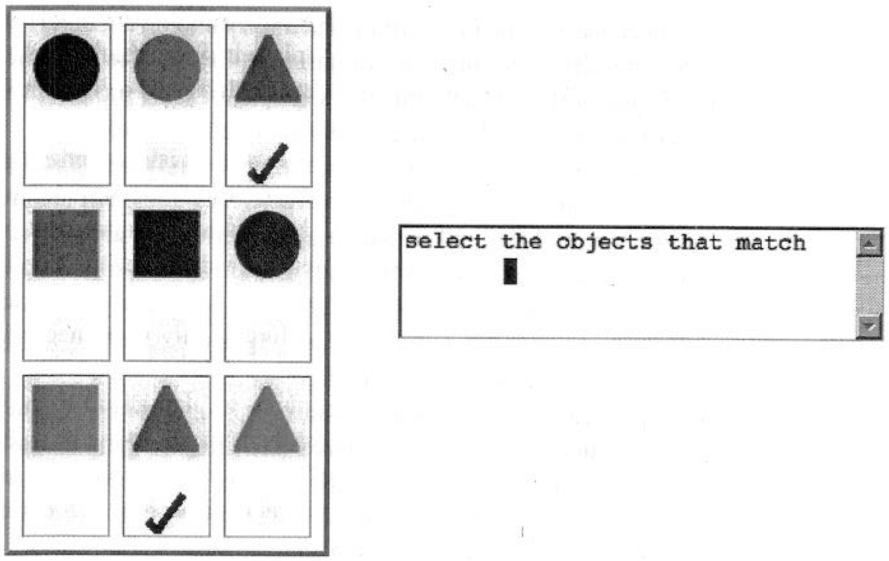


Figure 3.6. Sample problem from the study. In this problem, the participant is asked to write a textual query to select the objects that are marked. The color of each object is red, green, or blue on the computer screen.

marked and they had to formulate a query to select them; and code interpretation problems, where they were shown a query and had to mark the shapes selected by the query. They solved all of these problems twice, once using purely textual queries, and once using match forms.

The results suggest that a tabular language for specifying Boolean expressions can improve the usability of a programming or query language. On code generation tasks, the participants performed significantly better using the tabular form, while on code interpretation tasks they performed about equally in the textual and tabular conditions. The study also uncovered systematic patterns in the ways participants interpreted Boolean expressions, which contradict the typical rules of evaluation used by programming languages. These observations help to explain some of the underlying reasons why Boolean expressions are so difficult for people to use accurately, and suggest that refining the vocabulary and rules of evaluation might improve the learnability and usability of textual query languages. A general awareness of these contradictions can help designers of future query systems adhere to the HCI principle to *speak the user's language* (Nielsen, 1994). Additional details about this study were reported by Pane and Myers (2000).

6. Hands Environment and Language

The next step was to design and implement HANDS, our new programming language and environment. The various components of this system were designed in response to the observations in our studies as well as prior work:

- Beginners have difficulty learning and understanding the highly-detailed, abstract, and unfamiliar concepts that are introduced to explain how most programming languages work. HANDS provides a simple concrete model based on the familiar idea of a character sitting at a table, manipulating cards.
- Beginners have trouble remembering the names and types of variables, understanding their lifetimes and scope, and correctly managing their creation, initialization, destruction, and size, all of which are governed by abstract rules in most programming languages. In HANDS, all data are stored on cards, which are familiar, concrete, persistent, and visible. Cards can expand to accommodate any size of data, storage is always initialized, and types are enforced only when necessary, such as when performing arithmetic.
- Most programming languages require the programmer to plan ahead to create, maintain, and traverse data structures that will give them access to the program's data. Beginners do not anticipate the need for these structures, and instead prefer to access their data through content-based retrieval as needed. HANDS directly supports queries for content-based data retrieval.
- Most programming languages require the programmer to use iteration when performing operations on a group of objects. However, the details of iteration are difficult for beginners to implement correctly, and furthermore, beginners prefer to operate on groups of objects in aggregate instead of using iteration. HANDS uniformly permits all operations that can be performed on single objects to also be performed on lists of objects, including the lists returned by queries.
- Despite a widespread expectation that visual languages should be easier to use than textual languages, the prior work finds many situations where the opposite is true (Blackwell, 1996; Green and Petre, 1992; Whitley, 1997). In our studies, pictures were often used to describe setup information, but then text was used to describe dynamic behaviors. HANDS supports this hybrid approach, by permitting objects to be created and set up by direct manipulation but requiring most behaviors to be specified with a textual language. This design assumes that the environment will provide syntax coloring and other assistance with syntax. These features are commonly available in programming environments, but re-implementing them in HANDS was beyond the scope of our work.
- Programming language syntax is often unnatural, laden with unusual punctuation, and in conflict with expectations people bring from their knowledge in other domains such as mathematics. The HANDS language minimizes punctuation and has a more natural syntax that is modeled after the language used by non-programmers in our studies.
- The prior research offers few recommendations about which programming paradigm might be most effective for beginners (imperative, declarative, functional, event-based, object-oriented, etc.). In our studies of the natural ways beginners expressed problem solutions, an event-based paradigm was observed most often, and program entities were often treated with some object oriented features. HANDS therefore uses an event-based paradigm. Cards are the primary data

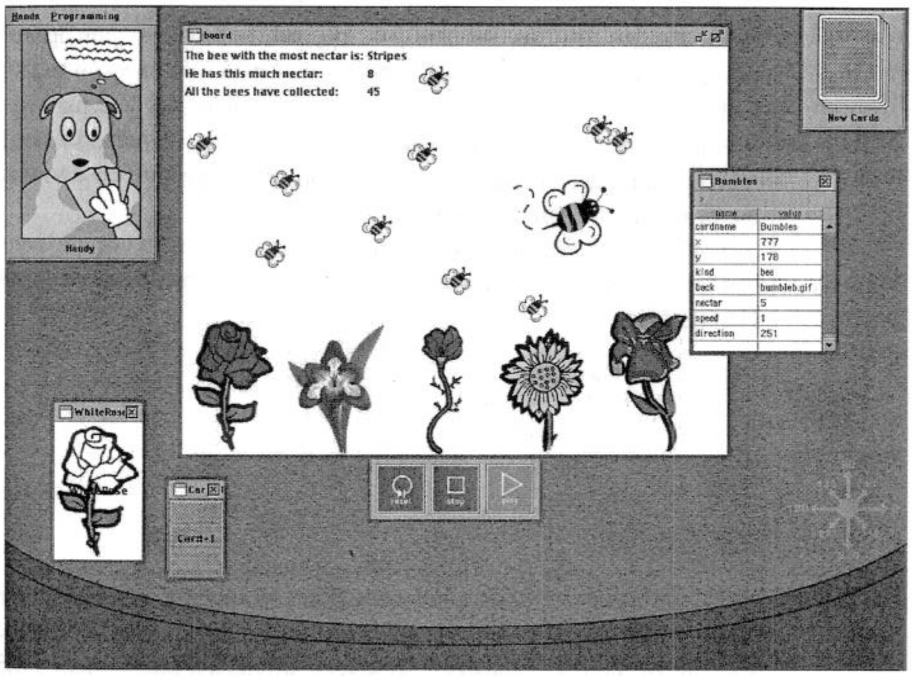
structure, and they have some object-like properties: they are global, named, encapsulated, persistent, and have some autonomous behaviors.

- The prior work recommends that programming systems should provide high-level support for the kinds of programs people will build, so they do not have to assemble more primitive features to accomplish their goals. In our interviews with children, they said they wanted to create interactive graphical programs like the games and simulations they use every day. HANDS provides domain-specific support for this kind of program.

All of these observations have influenced the design of HANDS. 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.

6.1. COMPUTATIONAL MODEL

In HANDS, the computation is represented as an agent named Handy, sitting at a table manipulating a set of cards (see Figure 3.7). All of the data in the system is stored on



```
The bee with the most nectar is: Stripes
He has this much nectar:         8
All the bees have collected:     45
```

name	value
cardname	Bumbles
x	777
y	178
kind	bee
back	bumbleb.gif
nectar	5
speed	1
direction	251

Figure 3.7. The HANDS system portrays the components of a program on a round table. All data are 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.

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.[1]

6.2. PROGRAMMING STYLE AND MODEL OF EXECUTION

HANDS is event-based, the programming style that most closely matches the problem solutions in our 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 that Handy should execute in response to the event. For example, this event handler would be run if the system detects a collision between a bee and flower, changing the *speed* value on the bee's card:

```
when any bee collides into any flower
      set the bee's speed to 0
end when
```

6.3. AGGREGATE OPERATIONS

In our studies, we 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.

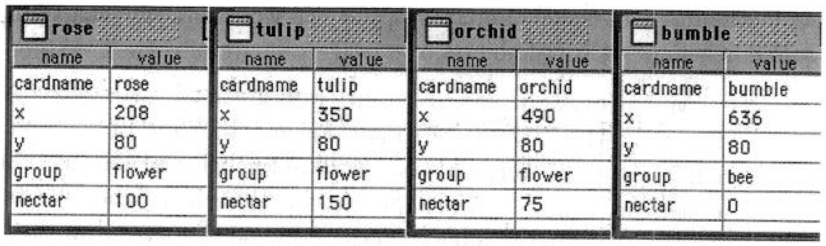
HANDS has full support for aggregate operations. All operators can accept lists as well as singletons as operands. For example, all of the following are legal operations in HANDS:

```
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
```

6.4. QUERIES

In our studies, we 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.

[1] HANK (Mulholland & Watt, 2000) is another end-user programming system where the program is represented as a set of cards and an agent was represented by a cartoon dog in early prototypes. The resemblance of HANDS to HANK is coincidental.



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 3.8. When the system evaluates the query `all flowers` it returns `orchid, rose, tulip`.

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 3.8 contains cards representing three flowers and a bee to help illustrate the following queries.

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

6.5. 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.

7. Evaluation of the Hands Environment and Language

7.1. USER STUDY

To examine the effectiveness of three key features of HANDS—queries, aggregate operations, and data visibility—we 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. For example, in this limited version aggregate operations were not available, so iteration was required to act upon a list of objects.

Volunteers were recruited from a fifth-grade class at a public school to participate in the study. The 23 volunteers ranged in age from 9 to 11 years old. There were 12 girls and 11 boys. All were native speakers of English, and none had computer programming experience. They came to a university campus on a Saturday morning for a three-hour session. 12 of the children used HANDS and 11 used the limited version of the system.

The HANDS users learned the system by working through a 13-page tutorial. A tutorial for the limited-feature version was identical except where it described a feature that was missing in the limited system. In those places, the tutorial taught the easiest way to accomplish the same effect using features that remained. These changes increased the length of the tutorial slightly, to 14 pages. After working through the tutorial, the children attempted to solve five programming tasks.

In this three-hour session, the children using HANDS were able to learn the system, 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. The HANDS users solved an average of 2.1 programming problems, while the children using the limited version were able to solve an average of 0.1 problems ($p <.05$). This provides evidence that the three key features improve usability over the typical set of features in programming systems. Additional details about this study were reported by Pane and Myers (2002).

7.2. DISCUSSION

The ease with which these children were able to learn the system and use it to solve tasks suggests that HANDS is a gentle slope system; or, at least its curve has a gentle slope near the origin of the sophistication-difficulty chart (Figure 3.1). The system has a broad range of capabilities. Adults have used it to create interactive games and scientific simulations, as well as solutions to classical computer science problems such as *Towers of Hanoi* and the computation of prime numbers. However, additional testing is necessary to see how far the gentle slope persists, and, if there are any walls, where and how high they are.

Although HANDS was designed for children, we expect that many of its features are generally useful for end-user developers of all ages. This is because most of the factors that drove the design of HANDS (see section 6) were general to beginners and not specific to children. Anecdotally, several of the features of HANDS are attractive to highly experienced programmers; however we have not gathered any empirical evidence on whether this design is generally suitable for programmers across all levels of experience.

HANDS was designed to support the domain of highly interactive graphical programs. HANDS is not well suited to problems outside this domain, such as word processing, technical drawing, or web browsing. It is lacking domain-specific features such as text layout support; and features of HANDS such as collision detection may have little use in these other domains. However, the natural programming design process could be used to design other systems to support any of these domains. Since most of the underlying computational features, such as queries and aggregate operations, are not domain-specific, they are likely to be important features of these other systems as well.

8. Current Work

Understanding of code and debugging are significant areas of difficulty for novices (du Boulay, 1989), but somewhat surprisingly, there has been little advancement in the tools to help with these problems. Even environments aimed at novices have few facilities to help with debugging. For example, systems such as MacroMedia's Director and Flash, Microsoft's Visual Basic, and general-purpose programming environments like MetroWerks' CodeWarrior and Microsoft's Visual C++, all provide basically the same debugging techniques that have been available for 60 years: breakpoints, print statements and showing the values of variables. In contrast, the questions that programmers need to answer are at a much higher level. Our current work is investigating what questions are the most *natural* for users to ask when they are trying to understand and debug their code. Our initial user studies (Ko and Myers, 2003) have shown that often, users are trying to find the answers to "why" and "why not" questions such as:

- Why did the object become invisible?
- Why does nothing happen when I click on this button?
- What happened to my graphical object?
- Where did this value get set?

We are now developing new tools that will allow users to ask such questions directly in the programming environment while debugging. We are conducting user studies to evaluate to what extent the tools enable users to ask questions in a natural way, and to determine what kinds of code and data visualizations will provide the most helpful answers (Ko and Myers, 2004).

9. Conclusions

While making programming languages and environments more natural may be controversial when aimed at professional programmers, it has significant importance for end-user development. In addition to supplying new knowledge and tools directly, the human-centered approach followed by the Natural Programming project provides a methodology that can be followed by other developers and researchers when designing their own languages and environments. We believe this will result in more usable and effective tools that allow both end-users and professionals to write more useful and correct programs.

Acknowledgments

This research has been funded in part by the National Science Foundation under grants IRI-9900452 and IIS-0329090, and as part of the EUSES Consortium under grant ITR-0325273. 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. and Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human–Computer Interaction*, **1**, 107–131.

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? In: *Proceedings of the VL'96 IEEE Symposium on Visual Languages*. Boulder, CO: IEEE Computer Society Press, pp. 240–246.

Bonar, J. and Cunningham, R. (1988). Bridge: Tutoring the programming process. In: J. Psotka, L.D. Massey and S.A. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 409–434.

Bonar, J. and Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 325–353.

Davies, S.P. (1993). Externalising information during coding activities: Effects of expertise, environment and task. In: C.R. Cook, J.C. Scholtz and J.C. Spohrer (eds.), *Empirical Studies of Programmers: Fifth Workshop*. Palo Alto, CA: Ablex Publishing Corporation, pp. 42–61.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 283–299.

du Boulay, B., O'Shea, T. and Monk, J. (1989). The black box inside the glass box: Presenting computing concepts to novices. In E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 431–446.

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.

Green, T.R.G. and Petre, M. (1992). When visual programs are harder to read than textual programs. In: G.C. van der Veer, M.J. Tauber, S. Bagnarola and 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. 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.

Hildreth, C. (1988). Intelligent interfaces and retrieval methods for subject search in bibliographic retrieval systems. In *Research, Education, Analysis and 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? In: E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 179–190.

Hoc, J.-M. and Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In J.-M. Hoc, T.R.G. Green, R. Samurçay and D.J. Gilmore (eds.), *Psychology of Programming*. London: Academic Press, pp. 139–156.

Hutchins, E.L., Hollan, J.D. and Norman, D.A. (1986). Direct manipulation interfaces. In D.A. Norman and S.W. Draper (eds.), *User Centered System Design: New Perspectives on Human–Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Ko, A.J. and Myers, B.A. (2003). Development and evaluation of a model of programming errors. In: *Proceedings of IEEE Symposia on Human-Centric Computing Languages and Environments*. Auckland, New Zealand. pp. 7–14.

Ko, A.J. and Myers, B.A. (2004). Designing the Whyline: A debugging interface for asking questions about program behavior. In Proceedings of CHI 2004 Conference on Human Factors in Computing Systems. Vienna, Austria: ACM Press, pp. 151–158.

MacLean, A., Carter, K., Lövstrand, L. and Moran, T.P. (1990). User-tailorable systems: Pressing the issues with buttons. In J.C. Chew and J. Whiteside (eds.), *Proceedings of CHI'90 Conference on Human Factors in Computing Systems*. Seattle, WA: ACM Press, pp. 175–182.

Mayer, R.E. (1989). The psychology of how novices learn computer programming. In E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 129–159.

McQuire, A. and Eastman, C.M. (1995). Ambiguity of negation in natural language queries. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. Seattle, WA: ACM Press, p. 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.

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.

Myers, B.A., Smith, D.C. and Horn, B. (1992). Report of the 'End User Programming' working group. In *Languages for Developing User Interfaces*. Boston, MA: Jones and Bartlett.

Nielsen, J. (1993). *Usability Engineering*. Chestnut Hill, MA: AP Professional.

Nielsen, J. (1994). Heuristic evaluation. In J. Nielsen and R.L. Mack (eds.), *Usability Inspection Methods*. New York: John Wiley & Sons, pp. 25–62.

Norman, D.A. (1986). Cognitive engineering. In D.A. Norman and S.W. Draper (eds.), *User Centered System Design: New Perspectives on Human–Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Pane, J.F. (2002). *A Programming System for Children that is Designed for Usability.* Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA.

Pane, J.F. and Myers, B.A. (2000). Tabular and textual methods for selecting objects from a group. In *Proceedings of VL 2000: IEEE International Symposium on Visual Languages*. Seattle, WA: IEEE Computer Society, pp. 157–164.

Pane, J.F. and Myers, B.A. (2002). The impact of human-centered features on the usability of a programming system for children. In *CHI 2002 Extended Abstracts: Conference on Human Factors in Computing Systems*. Minneapolis, MN: ACM Press, pp. 684–685.

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.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *IEEE Computer*, **16**(8), 57–69.

Smith, D.C., Cypher, A. and Schmucker, K. (1996). Making programming easier for children. *Interactions* **3**(5), 59–67.

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. (1989). Cognitive strategies and looping constructs: An empirical study. In: E. Soloway and J.C. Spohrer (eds.), *Studying the Novice Programmer*. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 191–207.

Stein, L.A. (1999). Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems* **30**(6), 473–507.

Whitley, K.N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, **8**(1), 109–142.