Int. J. Human-Computer Studies (2001) **54,** 237–264
doi:10.1006/ijhc.2000.0410
Available online at http://www.idealibrary.com on IDEAL®

Studying the language and structure in non-programmers' solutions to programming problems

John F. Pane, Chotirat "Ann" Ratanamahatana and Brad A. Myers
Computer Science Department and Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA.
email: pane+ijhcs@cs.cmu.edu; http://www.cs.cmu.edu/~NatProg.

Programming may be more difficult than necessary because it requires solutions to be expressed in ways that are not familiar or natural for beginners. To identify what is natural, this article examines the ways that non-programmers express solutions to problems that were chosen to be representative of common programming tasks. The vocabulary and structure in these solutions is compared with the vocabulary and structure in modern programming languages, to identify the features and paradigms that seem to match these natural tendencies as well as those that do not. This information can be used by the designers of future programming languages to guide the selection and generation of language features. This design technique can result in languages that are easier to learn and use, because the languages will better match beginners' existing problem-solving abilities.

1. Introduction

Programming is a very difficult activity. Some of the difficulty is intrinsic to programming, but this research is based on the observation that programming languages make the task more difficult than necessary because they have been designed without careful attention to human–computer interaction issues (Newell & Card, 1985). In particular, programmers are 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; in contrast, this can be done in a spreadsheet with a single line of code using the *sum* operator (Green & Petre, 1996). The mismatch between the way programmers think 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.

The *Natural Programming Project* seeks to influence future programming languages by collecting a human-centered body of facts that can be used to guide design decisions (Myers, 1998). This will allow the language designer to be more aware of the areas of potential difficulty, as well as suggest alternate approaches that more closely match the ways that people think. This article describes a pair of new studies that examine the language and structure of problem solutions written by non-programmers, and contrasts

1071-5819/01/020237 + 28 $35.00/0

these findings with the requirements imposed by popular modern programming languages.

The first study focuses on children because they are the audience for a new programming language the authors are designing. In addition, children are less likely to be programmers, so their responses should reveal problem-solving techniques that have not been influenced by programming experience. The exercises in this study are drawn from the domain of computer games and animated stories, because children are often interested in building these kinds of programs. The second study then examines how the results of the first study generalize to a broader audience and a different domain.

2. Related work

There has been a wealth of relevant research in the fields of *Psychology of Programming* and *Empirical Studies of Programmers*. In these fields, programming is often defined as a process of transforming a mental plan that is in familiar terms into one that is compatible with the computer (e.g. Lewis & Olson, 1987). Among others, Hoc and Nguyen-Xuan (1990) have shown that many bugs and difficulties arise because the distance between these is too large. This concept is called *closeness of mapping* by Green & Petre (1996, p. 146): "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." In that article, they provide an extensive set of *cognitive dimensions*, which can be used to guide and evaluate language designs. More concretely, Soloway, Bonar and Ehrlich (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 & Soloway, 1989). For example, *then* is interpreted as *afterwards* instead of *in these conditions*. Many similar findings are summarized in an earlier report (Pane & Myers, 1996). While these studies identify many of the problems with existing languages, they do not prescribe solutions. The goal of the current work is to discover alternatives that can avoid or overcome these problems.

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 of the problems that are listed above, as well as other usability problems. For example, Thimbleby, Cockburn and Jones (1992) list many ways that HyperTalk violates the human–computer interaction principle of consistency. 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, Bruckman and Edwards (1999) have found that leveraging users' natural-language-like knowledge in a more formalized syntax is an effective strategy for designing end-user programming languages.

There are many motivations for why a more natural programming language might be better. 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 and Norman (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 been a consideration in most programming language designs.

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* (1993, p. 38), 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 require the programmer to make tremendous transformations from the intended tasks to the code design.

The current studies are similar to a series of studies by Lance Miller (1974, 1981) in the 1970s. Miller 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 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, Ballard and Sigmon (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 not had any direct impact on the designs of new programming languages that have been invented since then.

The studies reported in this article differ from the prior art in several ways. For example,

- Miller's studies used verbose problem statements, raising the risk that the language used in the participants' responses was biased by the materials. In fact, one of the

frequently observed keywords actually appeared in the problem statement that was given to the participants. The current studies take great care to minimize this kind of bias by using terse descriptions along with graphical depictions of the problem scenarios.

- Miller's studies placed constraints on the participants' solutions, such as: they were broken into *steps*, each line of the text is limited to 80 characters; steps had to be retyped completely in order to edit them; and, a minimum of five steps was required in a solution. The current studies are much less constrained, allowing users to write or draw as much or as little text and pictures as they need to convey their solutions.
- Miller's tasks were typical database problems from the era of his studies. The current studies investigate a broader range of tasks that incorporate modern graphical user interfaces and media such as animations.
- Miller's participants were all college students. The current studies investigate a broader age range.

Thus, the current studies may yield more reliable information about the natural expressions of a wider audience, on a broader range of algorithms and domains.

3. The studies

In the studies reported here, 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, the authors enumerated a list of essential programming techniques and concepts that are needed to program various kinds of applications. These include: use of variables, assignment of values, initialization, comparison of values and 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. The authors chose the PacMan video game 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 most of the concepts in a limited amount of time. The skills that were not covered in the first study were covered in the second, which used scenarios that involved database manipulation and numeric computation.

A risk in designing these studies is that the experimenter could bias the participants by 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. To avoid this, a collection of pictures and QuickTime movie clips were developed to depict the various scenarios, using very terse captions. This enabled the experimenter to show the depictions and ask vague questions to prompt the participants for their responses. An example is shown in Figure 1. Full details about these studies,

including copies of the materials and complete tabulation of the results are available in a supplementary report (Pane, Ratanamahatana & Myers, 2000).

4. Study one

The first study examines children's solutions to a set of tasks that would be necessary to program a computer game.

4.1. PARTICIPANTS

Fourteen fifth graders at a Pittsburgh public elementary school participated in this study. The participants were equally divided between boys and girls, were racially diverse and were either 10 or 11 years old. All of the participants were experienced computer users, but only two of them (both boys) said they had programmed before. All of the analyses in this article examine only the 12 non-programmers. The participants were recruited by sending a brief note and consent form to parents. The participants received no reward other than the opportunity to leave their normal classroom for half an hour and the opportunity to play a computer game for a few minutes.

4.2. MATERIALS

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, 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 1 shows one of the scenario depictions. The participants viewed the depictions on a color laptop computer and wrote their solutions on blank unlined paper.

4.3. PROCEDURE

After a brief interview to gather background information, participants were shown each scenario and asked to write down in their own words and pictures how they would tell the computer to accomplish the scenario. When a response was judged to be incomplete or unsatisfactory, the experimenter attempted to elicit additional information by asking the participant to give more detail, by demonstrating an error in the existing answer, or by asking questions that were carefully worded to avoid influencing the responses. The sessions were audiotaped.

4.4. CONTENT ANALYSIS

The authors developed a rating form to be used by independent raters to analyse each participant's responses. 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

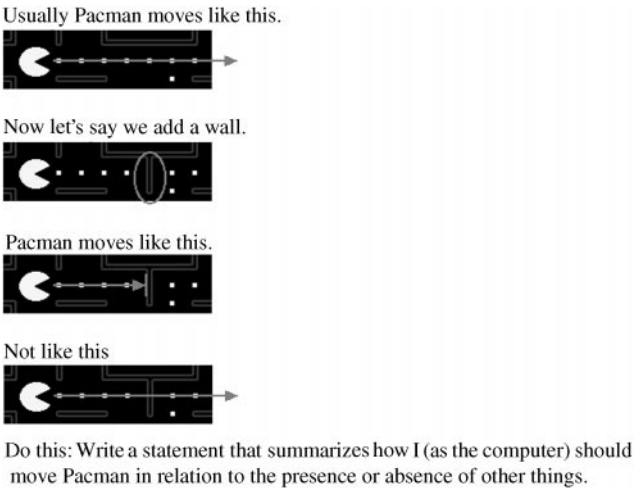

FIGURE 1. Depiction of a problem scenario in study one.

some other characteristic of the language or strategy that was employed. Many of these questions arose from the results of a pilot study. In addition, preliminary review of the participant data revealed trends in the solutions that the authors thought were important, so the rating form was supplemented with questions to explore these as well.

Each question was followed by several categories into which the participant's responses could be classified. The rater was instructed to look for relevant sentences in the participant's solution, and classify each one by placing a tickmark 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.

Five independent raters categorized the participants' responses. These raters were experienced computer programmers, who were recruited by posting to Carnegie Mellon University's electronic bulletin boards and were paid for their assistance. They were given a one-page instruction sheet describing their task. Each analyst filled out a copy of the 17-question rating form for each of the participants. Figure 2 shows one of the questions from the rating form for study one.

4.5. RESULTS

The participants' solutions ranged from one to seven pages of handwritten text and drawings. The raters were instructed to use each utterance (statement or sentence) as the unit of text to analyse. Since each rater independently partitioned the text into these units, the total number of tickmarks differed across raters, so the results are normalized by looking at the proportion of the tickmarks credited to each category rather than the raw counts. Although there were variances among the results from individual raters,

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 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 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 2. A question from the rating form for study one.

their ratings were generally similar. So the results are reported as averages across all raters ($n = 5$) and all the non-programmer participants ($n = 12$).

The results for each rating form question are summarized with an overall *prevalence* score followed by *frequency* scores for each category sorted from most frequent to least frequent. The prevalence score measures the average count of occurrences that each rater classified for each participant when answering the current question. In study one, this score varies from 1.0 to 23.2, indicating the relative amount of data that was available to the raters in answering the question. The frequency scores then show how those occurrences were apportioned across the various categories, expressed as percentages. The frequencies may not sum to exactly 100% due to rounding errors. The examples are quoted from the participants' solutions. Table 1 summarizes the results that follow, which are sorted into four general categories: the overall structure of the solutions, the ways that certain keywords are used, the kinds of control structures that are used and the methods used to effect various aspects of computation.

4.6. OVERALL STRUCTURE

4.6.1. Programming style. The raters classified each statement or sentence in the solutions into one of the following categories based on the style of programming that it most closely matches.
Prevalence: 22.7 occurrences per participant.

- 54%—production rules or event-based, beginning with *when*, *if* or *after*.
  Example: *When PacMan eats all the dots, he goes to the next level.*
- 18%—constraints, where relations are stated which should always hold.
  Example: PacMan cannot go through a wall.
- 16%—other (98% of these were classified by the raters as declarative statements).
  Example: *There are* 4 *monsters.*
- 12%—imperative, where a sequence of commands is specified.
  Example: *Start with this image. Play this sound. Display* "*Player One Get Ready*."

TABLE 1

Summary of results from the study. Items with frequencies below 5% do not appear

Overall structure

Programming style
54% Production rules/events
18% Constraints
16% Other (declarative)
12% Imperative

Perspective
45% Player or end-user
34% Programmer
20% Other (third-person)

Modifying state
61% Behaviors built into objects
20% Direct modification
18% Other

Pictures
67% Yes

Keywords

AND
67% Boolean conjunction
29% Sequencing

OR
63% Boolean disjunction
24% To clarify or restate a prior item
8% "Otherwise"
5% Other

THEN
66% Sequencing
32% "Consequently" or "in that case"

Control structures

Operations on multiple objects
95% Set/subset specification
5% Loops or iteration

Complex conditionals
37% Set of mutually exclusive rules
27% General case, with exceptions
23% Complex boolean expression
14% Other (additional uses of exceptions)

Looping constructs
73% Implicit
20% Explicit
7% Other

Computation

Remembering state
56% Present tense for past event
19% "After"
11% State variable
6% Discuss future events
5% Past tense for past event

Tracking progress
85% Implicit
14% Maintain a state

Mathematical operations
59% Natural language style — incomplete
40% Natural language style — complete

Motions
97% Expect continuous motion

Randomness
47% Precision
20% Uncertainty without using "random"
18% Precision with hedging
15% Other

Insertion into a data structure
48% Insert first then reposition others
26% Insert without making space
17% Make space then insert
8% Other

Sorted insertion
43% Incorrect method
28% Correct non-general method
18% Correct general method

4.6.2. Perspective. Beginners sometimes confuse their role or perspective while they are developing a program. Instead of thinking about the program from the perspective of the programmer, they might adopt the role of the end-user of the program, or in the case of games and stories, one of the characters portrayed by the program. The raters classified the participants' statements according to the perspective or role that they indicated.
Prevalence: 23.2 occurrences per participant.

- 45%—player's or end-user's perspective.
  Example: *When I push the left arrow PacMan goes left.*
- 34%—programmer's perspective.
  Example: *If arrow for Player 1 is "left" move PacMan left.*
- 20%—other (99% of these were classified by the raters as *third-person perspective*).
  Example: *If he eats a power pill and he eats the ghosts, they will die.*

4.6.3. Modifying State. The raters examined places where the participants were making changes to an entity.
Prevalence: 4.6 occurrences per participant.

- 61%—behaviors were built into the entity, in an object-oriented fashion.
  Example. Get the big dot and the ghost will turn colors …
- 20%—direct modification of the properties of entities.
  Example: *After eating a large dot, change the ghosts from original color to blue.*
- 18%—other.

4.6.4. Pictures. In addition to the above classifications done by the raters, the experimenter examined each solution to determine whether pictures were drawn as part of the solution.

- 67%—included at least one picture.
- 33%—used text only.

4.7. KEYWORDS

4.7.1. AND. The raters examined the intended meaning when the participants used the word *AND*.
Prevalence: 6.3 occurrences per participant.

- 67%—boolean conjunction.
  Example: *If PacMan is travelling up and hits a wall, the player should …*
- 29%—for sequencing, to mean *next* or *afterward*.
  Example: *PacMan eats a big blinking dot, and then the ghosts turn blue.*
- 3%—other
  Example: *Every level the fruit should stay for less and less seconds.*

4.7.2. OR. The raters examined the intended meaning when the participants used the word *OR*.
Prevalence: 1.5 occurrences per participant.

- 63%—boolean disjunction.
  Example: *To make PacMan go up or down, you push the up or down arrow key.*
- 24%—clarifying or restating the prior item.
  Example: *When PacMan hits a ghost or a monster, he loses his life.*
- 8%—meaning *otherwise.*
- 5%—other.

4.7.3. THEN. The raters examined the intended meaning when the participants used the word *THEN*.
Prevalence: 2.2 occurrences per participant.

- 66%—sequencing, to mean *next* or *afterward.*
  Example: *First he eats the fruit, then his score goes up 100 points.*
- 32%—meaning *consequently or in that case.*
  Example: *If you eat all the dots then you go to a higher level.*
- 1%—to mean *besides* or *also.*
- 1%—other.

4.8. CONTROL STRUCTURES

4.8.1. Operations on multiple objects. The raters examined those statements that operate on multiple objects, where some or all of the objects are affected by the operation.
Prevalence: 6.1 occurrences per participant.

- 95%—set and subset specifications.
  Example: *When PacMan gets all the dots, he goes to the next level.*
- 5%—loops or iteration.
  Example: *# 5 moves down to # 6, # 6 moves to # 7, etc., until # 10 which is kicked off the high score list.*

4.8.2. Iteration or looping constructs. The raters examined those statements that were either implicit or explicit looping constructs.
Prevalence: 1.6 occurrences per participant.

- 73%—implicit, where only a terminating condition is specified.
  Example: *Make PacMan go left until a dead end.*
- 20%—explicit, with keywords such as *repeat*, *while and so on*, etc.
- 7%—other.

4.8.3. ELSE or equivalent clauses. The raters looked for occurrences of *ELSE* clauses or equivalent constructs in the participants' solutions. They simply counted these, without classifying them further.
Prevalence: 0.4 occurrences per participant.

4.8.4. Complex conditionals. The raters examined those statements that specify conditions with multiple options.

Prevalence: 2.3 occurrences per participant.

- 37%—a set of mutually exclusive rules.
  Example: *When the monster is green he can kill PacMan. When the monster is blue PacMan can eat the monster.*
- 27%—a general condition, subsequently modified with exceptions.
  Example: *When you encounter a ghost, the ghost should kill you. But if you have a power pill you can eat them.*
- 23%—Boolean expressions.
  Example: *After eating a blinking dot and eating a blue and blinking ghost, he should get points.*
- 14%—other (95% of these either listed the exception first or did not list a general case).
  Example: *If he gets a [power pill] then if you run into them you get points.*

4.9. COMPUTATION

4.9.1. Remembering state. The raters examined the methods used to keep track of state when an action in the past should affect a subsequent action.

Prevalence: 4.1 occurrences per participant.

- 56%—using present tense when mentioning the past event.
  Example: *When PacMan eats a special dot he is able to eat the ghosts.*
- 19%—using the word *after*.
  Example: *After using up the power pill, the ghosts can eat PacMan again.*
- 11%—using a state variable to track information about the past event.
  Example: *When the monster is blue PacMan can eat the monster.*
- 6%—mentioning the future event at the time of past event.
  Example: *When PacMan gets a shiny dot, then if you run into the ghosts, you get points.*
- 5%—using the past tense when mentioning the past event.
  Example: *In about 10 s, if PacMan didn't eat it take it off again.*
- 4%—other.

4.9.2. Tracking progress. The raters examined the methods used to keep track of progress through a long task.

Prevalence: 2.0 occurrences per participant.

- 85%—all or nothing, where tracking is implicit or done with sets.
  Example: *When PacMan gets all the dots, he goes to the next level.*
- 14%—using counting, where a variable such as a counter tracks the progress.
  Example: *When PacMan loses 3 lives, it's game over.*
- 1%—other.

4.9.3. Mathematical operations. The raters examined the kinds of notations used to specify mathematical operations.

Prevalence: 3.4 occurrences per participant.

- 59%—natural language style, missing the amount or the variable.
  Example: *When he eats the pill, he gets more points …*
- 40%—natural language style, with no missing information.
  Example: *When PacMan eats a big dot, add 100 points to the score.*
- 0%—programming language style (count = count + 20).
- 0%—mathematical style (count + 20).

4.9.4. Motions. The raters examined the participants' expectations about whether motions of objects should require explicit incremental updating.

Prevalence: 7.8 occurrences per participant.

- 97%—expect continuous motion, specifying only changes in motion.
  Example: *PacMan stops when he hits a wall.*
- 2%—continually update the positions of moving objects.
- 1%—other.

4.9.5. Randomness. The raters examined the methods used by the participants' in expressing events that were supposed to happen at uncertain times or with uncertain durations.

Prevalence: 1.4 occurrences per participant.

- 47%—using precision, where no element of uncertainty is expressed.
  Example: *Put the new fruit in every 30 s.*
- 20%—using words other than *random* to express the uncertainty.
  Example: *The fruit will go away after a while.*
- 18%—using precision with hedging to express uncertainty.
  Example: *After around 3 or 4 more seconds the fruit disappears.*
- 15%—other (often the action was tied to another event).
  Example: *Put a fruit on the screen when PacMan is running out of power.*
- 0%—used the word *random.*

4.9.6. Insertion into a data structure. The raters examined the methods used by the participants to insert an element into the middle of an existing sequence of elements.

Prevalence: 1.0 occurrences per participant.

- 48%—inserting first, repositioning other elements afterwards.
- 26%—no mention of making room for the inserted element.
- 17%—making space by repositioning others, then inserting the element.
- 8%—other.

4.9.7. Sorted insertion. The raters examined the methods used by the participants to determine the correct place to insert an element into a sorted list.

Prevalence: 1.1 occurrences per participant.

- 43%—using an incorrect method, with missing or incorrect details.
- 28%—a method that is correct for the current data, but not a correct general solution.
- 18%—a correct general method that would work for any data.
- 10%—other.

4.10. DISCUSSION

Combined discussion of the two studies appears in Section 6.

5. Study two

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 similar to the participants in study one.

5.1. PARTICIPANTS

Nineteen adults from the Carnegie Mellon University community, ranging in age from 18 to 34, participated in the study (10 men, 9 women). In addition, 22 fifth graders, ages 10 or 11, participated (13 boys, 9 girls). These fifth graders were recruited from the same Pittsburgh public elementary school as study one, but it was a new academic year so none of the participants from study one were involved in study two. The participants were racially diverse. Although the children spanned a range of academic abilities, all of the Carnegie Mellon participants had strong academic backgrounds.

Of the adults, only five had never programmed before (2 men, 3 women). Of the children, 14 said they had never programmed before (11 boys, 3 girls). There is reason to believe that some of the children who claimed to be programmers did not accurately answer this question because they did not really know what programming is. Nonetheless, only those participants who said they never programmed were included in the analysis that follows.

The adult participants were recruited by word of mouth and signed the usual human subject consent forms. The children were recruited by sending a brief note and consent form to parents. The adult participants received no reward for their participation; the children had an opportunity to leave their normal classroom for a half hour and were given a snack at the end of their participation.

5.2. MATERIALS

A set of 11 scenarios were created, representing a progression of problems that a programmer might encounter in the process of creating and manipulating a database of names and numeric values. These scenarios were chosen to cover some of the essential concepts of programming that were not addressed in study one and to further elucidate some of the results from that study. As in study one, graphical depictions of these scenarios were developed. In this case they contained before and after pictures of database values in a tabular layout, with graphical annotations highlighting the differences between the before and after pictures, along with a minimal amount of text that was carefully chosen to avoid biasing the participants" responses. The topics of the scenarios were: entering values into the correct rows of a table, adding certain values in each row to produce a column of sums, discarding the smallest or largest value from each row when calculating the sum, assigning nominal values to each row depending on textual attributes or numeric ranges, producing a numerically sorted summary table with

No.	First name	Last name	Average score	Performance
1	Sandra	Bullock	3000	
2	Bill	Clinton	60 000	
3	Cindy	Crawford	500	
4	Tom	Cruise	5000	
5	Bill	Gates	6000	
6	Whitney	Houston	4000	
7	Michael	Jordan	20 000	
8	Jay	Leno	50 000	
9	David	Lettermen	700	
10	Will	Smith	9000	

Question 5A

- Describe in detail what the computer should do to obtain these results.

No.	First name	Last name	Average score	Performance
1	Sandra	Bullock	3000	Fine
2	Bill	Clinton	60 000	Extraordinary
3	Cindy	Crawford	500	Poor
4	Tom	Cruise	5000	Fine
5	Bill	Gates	6000	Fine
6	Whitney	Houston	4000	Fine
7	Michael	Jordan	20 000	Extraordinary
8	Jay	Leno	50 000	Extraordinary
9	David	Lettermen	700	Poor
10	Will	Smith	9000	Poor

FIGURE 3. Depiction of a problem scenario in study two.

entries for only the rows with the highest sums, adding or subtracting a fixed value to every value in a column, deleting rows from the table or adding rows to it, and zeroing all the values in a column. Figure 3 shows one of the scenario depictions. The depictions were displayed to the participants on paper and they wrote their solutions directly on the problem pages.

5.3. PROCEDURE

The same procedure was used as in study one, except the sessions were not audiotaped.

5.4. CONTENT ANALYSIS

Once again a form was developed, similar to the one used in study one, so that independent raters could analyse the data. This rating form had 18 questions. Because the performance of the five analysts in the first study was satisfactory, there was general agreement among them and the task was very tedious, the authors decided that three analysts were sufficient for the second study. The analysts from the first study were permitted to return for this study because there was no reason to expect their prior

participation to have a material affect on the results. So three analysts from the prior study analysed the participants' responses in this study.

5.5. RESULTS

The participants answers typically consisted of one to five sentences in response to each of the 11 questions. Once again, there was general agreement among the raters. The performance of adults was generally similar to the performance of children. So, the results reported below are averages across the raters ($n = 3$) and all of the non-programmer participants ($n = 19$, 5 adults and 14 children).

As in study one, the results for each question are summarized with an overall *prevalence* score followed by *frequency* scores for each category. The prevalence score measures the average count of occurrences that each rater classified for each participant when answering the current question. In study two, this score varies from 0.2 to 11.5, indicating the relative amount of data that was available to the raters in answering the question. The frequency scores then show how those occurrences were apportioned across the various categories, expressed as percentages. The frequencies may not sum to exactly 100% due to rounding errors. The examples are quoted from the participants' solutions. Table 2 summarizes the results that follow, which are sorted into three general categories: the ways that certain keywords are used, the kinds of control structures that are used and the methods used to effect various aspects of computation.

5.6. KEYWORDS

5.6.1. AND. The raters examined the intended meaning when the participants used the word *AND*.

Prevalence: 6.1 occurrences per participant.

- 47%—boolean conjunction.
  Example: *Erase Bill Clinton and Jay Leno.*
- 43%—sequencing, meaning *next* or *afterward*.
  Example: *Crossed out the highest score and added the lower scores.*
- 5%—other.
- 4%—to specify a range.
  Example: *Fine is between 3000 and 20 000.*

5.6.2. AND as a Boolean operator. The raters examined the answers to two questions that were likely to elicit Boolean expressions. If the word *AND* appeared in a Boolean expression, the raters determined whether it was used correctly.

Prevalence: 0.6 occurrences per participant.

- 76%—incorrect, interpreting as Boolean conjunction would not give the intended result.
  Example: *Everybody whose name starts with the letter G and L would be in the black group.*
- 24%—correct, Boolean conjunction is intended meaning.
  Example: *If AvgScore* $\geqslant 1000$ *and* $< 10\,000$, *say Fine.*

TABLE 2

Summary of results from the second study. Items with frequencies below 5% do not appear

Keywords

AND
47% Boolean conjunction
43% Sequencing
5% Other

AND as a Boolean operator
76% Incorrect
24% Correct

OR
100% Boolean disjunction

NOT
100% Low precedence

BUT
92% To mean "except"
8% Other

THEN
91% Sequencing
7% "Consequently"

Control structures

Operations on multiple
97% Sets and subsets, including plurals

Complex conditionals
45% Set of mutually exclusive conditions
36% Dependent clause cannot stand alone
16% Nested conditions

Computation

Set construction
46% Plurals
18% "Each" or "every"
16% Naming a column of the table
14% "All"

Set manipulation
45% Set inverse
29% Set difference
22% Disjoint or mutually exclusive sets
5% Other

Complete specification of ranges
50% Correct
50% Incorrect

Specifying open intervals
35% "Above" is exclusive
22% "Above" is inclusive
22% Powers of ten
15% Other
5% Mathematical notation

Specifying closed intervals
35% "From … to" is inclusive
19% Powers of ten
10% Mathematical notation
9% Other
9% "Between" used inconsistently
7% "From … to" used inconsistently
6% "Between" is inclusive
5% Ends of interval specified separately

Mathematical operations
52% Natural language style — complete
40% Other

Sorting
37% "Alphabetical", etc.
36% "From A to Z", etc.
11% Concrete example
9% Provide a key to a sort operator

Deleting an element from a data structure
73% No hole expected after deletion
25% Repaired a hole after deletion

Inserting an element into a data structure
75% Insert without making space
16% Make space then insert
6% Insert then make space

Sorted insertion
46% Incorrect method
34% Correct non-general method
13% Correct general method
6% Insert then sort

5.6.3. OR. The raters examined the places where the participants used the word *OR* as a Boolean operator to see if it was used correctly.
Prevalence: 0.6 occurrences per participant.

- 100%—correct.
  Example: *[Score] in the hundreds or less is poor.*

5.6.4. NOT. The raters examined the places where the participants used the word *NOT* as a Boolean operator to see what operator precedence was intended.
Prevalence: 0.1 occurrences per participant.

- 100%—low precedence: *NOT A or B* means *NOT* (*A or B*).
  Example: *The Gold group [contains the people] with the first two letters in their last name that are not Le or Ga.*

5.6.5. BUT. The raters examined the intended meaning when the participants used the word *BUT*.
Prevalence: 0.2 occurrences per participant.

- 92%—to mean *except*.
  Example: *Add every element in the row, but the maximum.*
- 8%—other.
- 0%—to mean *and*.

5.6.6. THEN. The raters examined the intended meaning when the participants used the word *THEN*.
Prevalence: 1.3 occurrences per participant.

- 91%—sequencing, to mean *next* or *afterward*.
  Example: *Add up all the scores in each row, then subtract the lowest score in each row.*
- 7%—to mean *consequently* or *in that case*.
  Example: *If their name begins with a G or an L then put them in the Black group.*
- 1%—besides or also.
- 1%—other.

5.7. CONTROL STRUCTURES

5.7.1. Operations on multiple objects. The raters examined statements that operate on multiple objects, where some or all of the objects are affected by the operation.
Prevalence: 11.5 occurrences per participant.

- 97%—set or subset specifications, including the use of plurals.
  Example: *Select the four highest scores of the participants.*
- 3%—loop or iteration.
  Example: *Match the last name and fill the score until there is no more input.*
- 1%—other.

5.7.2. Complex conditionals. The raters examined statements specifying conditions with multiple options.

Prevalence: 1.3 occurrences per participant.

- 45%—a set of mutually exclusive conditions.
  Example: *If average score is less than 1000, performance is poor. If average score is between 1000 and 10 000, performance is fine. If average score is more than 10 000, performance is extraordinary.*
- 36%—a condition with a dependent clause that cannot stand alone.
  Example: *If the people's last name start with G or L they are on the black team. If not they are on the gold team.*
- 16%—nested conditions.
  Example: *If average score is in the hundreds it's poor. Less than 10 000 is fine.*
- 3%—other.

5.8. COMPUTATION

5.8.1. Set construction. The raters examined the places where sets are used, to determine how those sets were constructed.

Prevalence: 11.0 occurrences per participant.

- 46%—using plurals.
  Example: *Add the scores of 3 rounds.*
- 18%—using the words *each* or *every*.
  Example: *Add the score in every round.*
- 16%—naming a column of the table.
  Example: *Add 10 000 points to Round 1 and Round 3.*
- 14%—using the word *all*.
  Example: *Subtract [20 000 from] all elements in Round 2 …*
- 4%—enumerating the members of the set.
- 1%—other.

5.8.2. Set manipulation. The raters examined the ways that subsequent sets are created after an initial related set has been created.

Prevalence: 2.7 occurrences per participant.

- 45%—using set inverse, where the leftover items are operated on.
  Example: *If the last name begins with G or L, they are in the Black group. The rest are in the Gold group.*
- 29%—set difference, where some items are removed from the specified set.
  Example: *Add all the Rounds up except the highest score to get TOTAL.*
- 22%—constructing disjoint or mutually exclusive sets.
  Example: *Black is for G and L. Gold is for B, C, H, J and S.*
- 5%—other.

5.8.3. Complete specification of ranges. The raters examined the participants' statements that specify a range of integers, to see whether all of the possibilities were covered without holes or overlaps.

Prevalence: 1.3 occurrences per participant.

- 50%—correct.
  Example: *Scores below 1000 are poor. Scores from 1000 to 10 000 are fine. Any scores above 10 000 are extraordinary.*
- 50%—incorrect.

5.8.4. Specifying open intervals. The raters examined the participants' statements specifying open intervals, where all values beyond a single boundary are specified.

Prevalence: 2.0 occurrences per participant.

- 36%—words such as *above*, *below*, *greater than* or *less than* were intended to be exclusive.
  Example: *The performance of the person with the average scores below 1000 is considered as poor* (the participant then used *good* for 1000).
- 22%—words such as *above*, *below*, *greater than* or *less than* were intended to be inclusive.
  Example: *Poor would be below 999* (the participant then used *poor* for 999).
- 22%—powers of 10 were used to specify the range.
  Example: *If your score is in the hundred's your performance is poor.*
- 15%—other.
- 5%—mathematical notation, with inequality operators such as " $>$ " or " $\leqslant$ ".
  Example: *if score* < 1000, *performance* = *poor.*

5.8.5. Specifying closed intervals. The raters examined the participants' statements specifying closed intervals, where both boundaries are specified for a range of values.

1.2 occurrences per participant.

- 35%—*from … to*, the symbol "-", or similar notations are intended to be inclusive.
  Example: *The performance of ones whose average scores from 1000 up to 10 000 is considered as a fine performance* (the participant then assigned *fine* to both 1000 and 10 000).
- 19%—powers of 10 were used to specify the range.
  Example: *If your score is in the thousands, you are fine.*
- 10%—mathematical notation, with inequality operators such as " $>$ " or " $\leqslant$ ".
  Example: $1000 < x < 9999$; *performance* = *fine.*
- 9%—other.
- 9%—*between* is used with an inconsistent meaning at each end of the interval.
  Example: *If the average score is between 1000 and 10 000, the performance is fine* (the participant then assigned *fine* to 1000, and *extraordinary* to 10 000).
- 7%—*from … to*, the symbol "-", or similar notations are used with an inconsistent meaning at each end of the interval.
  Example: *Scores from* 1000 *to 10 000 are fine* (the participant then assigned *fine* to 1000, and *extraordinary* to 10 000).
- 6%—*between* is intended to be inclusive.
  Example: *Score between 1000 and 10 000 is fine* (the participant then assigned *fine* to both 1000 and 10 000).

- 5%—specified each end of the interval separately.
- 0%—*between* is intended to be exclusive.
- 0%—*from … to*, the symbol "-" or similar notations are intended to be exclusive.

5.8.6. Mathematical operations. The raters examined the kinds of notations used by the participants' in specifying mathematical operations.

Prevalence: 5.4 occurrences per participant.

- 52%—natural language style, with no missing information.
  Example: *Add 10 000 points to the scores in Rounds 1 and 3.*
- 40%—other (which includes natural language style, with missing amount or variable).
  Example: *Add up the scores of each person but don't add the highest number* (missing variable).
- 4%—mathematical notation.
  Example: *Column for r2 = x − 20 000.*
- 4%—programming language notation.

5.8.7. Sorting. The raters examined the participants' solutions to see how sorting operations were expressed.

Prealence: 1.3 occurrences per participant.

- 37%—using keywords such as *alphabetical* or *numerical.*
  Example: *Sort the table alphabetically.*
- 36%—using expressions like *from A to Z* or *from the lowest to the highest.*
  Example: *Put the 4 highest scores … in a different table from the highest to smallest.*
- 11%—using a concrete example from the current situation.
  Example: *Put him in number* 6 *because his last name comes before Jordan but after Houston.*
- 9%—using a sort key, such as *sort according to score.*
  Example: *Insert Elton John in order of the last name.*
- 4%—using words like *ascending* or *descending.*
  Example: *Sort* "*total score*" *column in descending order.*
- 4%—other.

5.8.8. Deleting an element from a data structure. The raters examined the methods used to delete an element from the middle of an existing sequence of elements, to see whether they expected a hole to be left behind.

Prevalence: 1.0 occurrences per participant.

- 73%—no hole was expected after the deletion.
  Example: *Take out Bill and Jay then put Elton John in.*
- 25%—fixed a hole after the deletion.
  Example: *Delete Row 2 and 8, moving everyone down to any unoccupied Rows.*
- 2%—other.

5.8.9. Insertion into a data structure. The raters examined the methods used to insert an element into the middle of an existing sequence of elements to see whether they expected that items would have to be arranged to make space for the new element.

Prevalence: 1.0 occurrences per participant.

- 75%—no mention of making room for the new element.
  Example: *Put Elton John in the records in alphabetical order.*
- 16%—make room for the element before inserting it.
  Example: *Use the cursor and push it down a little and then type Elton John in the free space.*
- 6%—make room for the element after inserting it.
- 4%—other.

5.8.10. Sorted insertion. The raters examined the methods used to determine the correct place to insert an element into a sorted sequence of elements.

Prevalence: 1.0 occurrences per participant.

- 46%—using an incorrect method, with missing or incorrect details.
  Example: *Insert row between number 5 and 7 and name it Elton John.*
- 34%—a method that is correct for the current data, but not a correct general solution.
  Example: *Put him in number 6 because his last name comes before Jordan but after Houston.*
- 13%—a correct general method that would work for all data.
  Example: *Insert Elton John into the table in alphabetical order of the last name.*
- 6%—insert then sort.
  Example: *Add Elton John, and then sort the table alphabetically.*
- 2%—other.

6. Discussion of results

This section contains discussion of the combined results from the two studies. In addition to interpretation of the results, this section includes some recommendations on how the programming system might be made more natural.†

6.1. PROGRAMMING STYLE

The majority of the statements written by the participants were in a production-rule or event-based style, beginning with words like *if* or *when*. However, the raters observed a significant number of statements using other styles, such as constraints, other declarative statements (that were not constraints) and imperative statements.

The dominance of rule- or event-based statements suggests that a primarily imperative language may not be the most natural choice. One characteristic of imperative languages is explicit control over program flow. Although imperative languages have *if* statements, they are evaluated only when the program flow reaches them. The participants' solutions seem to be more reactive, without attention to the global flow of control. When imperative statements were used, it was usually for local flow of control. The declarative

† These recommendations are not restricted to the programming *language* in isolation, but encompass the entire programming *system*, which includes the programming environment (editor, debugger, etc.), as well as the language. In modern programming systems these components all work in tandem, so it is useful to consider how the findings of this study might impact the entire system.

style seems to have been primarily used for setting up the scenario (data, characters, objects, etc.) of the program. Many of the constraints that were observed in this study were graphical in nature, such as objects that had certain fixed positions relative to one another or limitations on where those objects could go. The event-based style is used by several popular end-user programming environments such as Visual Basic, Lingo for Macromedia's Director and HyperTalk for HyperCard, although these systems have usability problems of their own (see, for example, Thimbleby *et al.*, 1992).

This mix of styles suggests that designers might be able to improve usability by not limiting the language to a single style. Different styles seem to be more natural for different parts of the programming task.

6.2. OPERATIONS ON MULTIPLE OBJECTS

The results from both studies highlight an important area where today's popular programming languages differ from the natural expressions used by the participants: the way that operations are performed on multiple objects. Most popular languages require iterative operation on the objects, one at a time, while the participants strongly preferred to use set and subset expressions or plurals, to specify the operations in aggregate. Miller (1974, 1981) made similar observations in his studies.

It has been well established in the literature that loops are a hotspot of difficulty and errors for novice programmers (du Boulay, 1989). And in many cases a loop is a more complicated and contorted way to specify operations that the participants were able to express easily and succinctly with aggregate set operations. New languages might support these aggregate operations, thus eliminating many of the cases where loops would otherwise be necessary.

Another requirement imposed by loops is the need to use extra variables to count iterations, flag terminating conditions or hold the current object being operated upon. This is even true in "high level" looping constructs such as *mapcar* in Lisp. The aggregate operations preferred by the participants reduce the need for these variables, which are another known area of difficulty for beginners (du Boulay, 1989). Spreadsheets provide a few aggregate operators, such as *sum*, but this feature is not generalized across all of the operators.

However, the participants did use looping constructs in a few cases, and the language should support these as well. Often, these loops use *until* to specify a terminating condition, while other times the terminating condition is implicit in phrases such as *and so on* or *etc*. In deciding the exact loop control structures to provide, the language designer should consider prior empirical studies which found that novices expect the terminating condition to be checked continuously, and the loop to halt the instant the condition is satisfied, rather than waiting until all of the subsequent statements inside the loop have been executed one last time (du Boulay, 1989).

6.3. SET CONSTRUCTION AND MANIPULATION

Study two illustrates a variety of ways that the participants construct sets: using plurals, the keywords *each*, *every* or *all* or by naming columns in a table. Once they had created a set, the participants often used operations such as inverse or difference to create

related sets. However, they sometimes preferred to create a separate disjoint set from scratch.

6.4. COMPLEX CONDITIONALS AND NOT

The participants used a number of ways to avoid writing complex Boolean conditionals. For example, they often wrote a series of mutually exclusive simple rules instead of a more complex conditional.

Also, they would sometimes express a general case followed by exceptions, as in:

```
if A do something unless B.
```

Notice that the equivalent Boolean expression that would be required to accomplish this in many programming languages involves not only a conjunction, but also the negation of the exception clause:

```
if A and not B do something.
```

The try … catch exception mechanisms in C++, Java, Lisp and other languages support this tendency by putting the general case first and listing the exceptions later, but other control structures in these languages do not. It might be useful to support the use of unless clauses throughout the language.

The raters found very few uses of negation. This is consistent with earlier findings that expressing negative concepts is more difficult than affirmative ones (Wason, 1959).

When the participants did use the *not* operator they gave it low precedence, which is contrary to the precedence that it has in most programming languages. A subsequent study found the use of *not* to be inconsistent: sometimes it was used with high precedence and other times with low precedence; and using parentheses was not effective to clarify precedence (Pane & Myers, 2000). Operator precedence errors were among the high-frequency bugs observed by Spohrer and Soloway (1986) in novice programs in a traditional programming language. However, in a recent study of a natural language style programming language, Bruckman and Edwards (1999) found that operator precedence errors were very infrequent. Further study is warranted to determine how languages should deal with issues of precedence.

6.5. MATHEMATICAL OPERATIONS

In study one, all of the mathematical operations were expressed in a natural language form; the raters found no mathematical or programming language notations. In study two, they found a very small amount of mathematical and programming notations among the adults' solutions. The vast preference for natural language mathematical operations should be supported by the programming language. However, more concise mathematical notation may be still necessary for calculations that are more complex than the ones required by the tasks in these studies.

Many of the mathematical expressions were missing either the variable on which to operate or the amount of the operation. This might be solved by providing slots that make the missing information more obvious, or by entering into a dialog with the user, with questions such as *how much*? or *to what*?

6.6. SPECIFICATIONS OF RANGES AND INTERVALS

In study two, the raters found that the participants were only about 50% successful in specifying ranges without holes or overlaps. Adults were more successful than children, possibly because they had mathematical notations for inequality in their arsenal. The children never made use of these mathematical notations. Instead they used powers of 10, or natural language expressions of inequality such as *above* or *greater than*. However, the participants were inconsistent about whether these latter terms were inclusive or exclusive. Adults achieved 100% accuracy when they used mathematical notations, suggesting that these are a better choice for audiences who understand them.

6.7. TACKING PROGRESS AND REMEMBERING STATE

The participants often avoided the use of variables to track progress in a task. This is not surprising because, as mentioned above, variables are an area of difficulty for novice programmers (du Boulay, 1989). Instead of variables, the participants preferred to use terms like *all* or *none* to detect when the task is finished. When they needed to use historical information to make decisions about present actions (or present information to make decisions about future actions), the participants usually did not use state variables to record the information. Instead they used future and past tenses to refer to the needed information. State variables are the only way to accomplish this in most programming languages. The challenge for language designers is to find ways to accommodate the more natural preferences.

6.8. AND, OR AND BUT

The raters found that often the word *and* was used as a sequencing word rather than as a Boolean operator. Also, in study two the raters examined the Boolean uses of *and*, and found that 75% were used in situations where the *or* operator would be required to achieve the desired effect in today's programming languages, as well as the query languages used for most database search engines. For example, a subject said, "if you score 90 and above", but the score cannot simultaneously be 90 *and* greater than 90. Because the natural uses of *and* have such diverse meanings, and most of them are inconsistent with the Boolean operator, designers of future language should consider substituting a different name or symbol for this operator.

Or and *but* appeared too rarely in these studies to draw firm conclusions without further research. When *or* was used, a Boolean interpretation would result in correct results. The infrequent use of *or* may be because disjunctive expressions are cognitively more difficult than conjunctive ones (Bourne, 1966).

6.9. THEN

The raters found that the most popular use of the word *then* is for sequencing or specifying that an action should happen after finishing a prior action. This is inconsistent with its use in most programming languages, where it means *consequently*. This confirms an earlier observation by du Boulay (1989).

6.10. DATA STRUCTURE OPERATIONS: INSERTION, DELETION, SORTING

When the participants were inserting and deleting data elements, they often did not consider issues about storage space that come up when working with the array data structures in most popular programming languages. This suggests that a built-in list-like data structure such as in the Lisp language, may be more natural.

The participants seemed to expect sorting to be a basic operator that they could utilize in their solutions, using expressions like *alphabetical* or *from A to Z*. When they were asked to provide an algorithm for sorting, they were rarely able to do this in a correct general way.

6.11. RANDOMNESS AND UNCERTAINTY

The raters did not find any uses of the word *random* in study one. Instead, the participants either expressed things with precision or used other ways of expressing uncertainty. Sometimes they tied the uncertain event to some other event that would happen at some unknown time. Perhaps the system could supply the uncertainty that is implicit in phrases like *about 3 s*.

6.12. OBJECT ORIENTED

Some aspects of object-oriented programming were apparent in the participants' solutions. Entities were treated as if they have state and an ability to respond to requests for action. However, there was no evidence in these studies of other aspects of object-oriented programming such as inheritance or polymorphism. Cypher and Smith (1995) found in user studies that inheritance hierarchies cause difficulty for children. Even among professional programmers, researchers have found that full-fledged object-oriented programming is not necessarily natural (Détienne, 1990; Glass, 1995).

6.13. MOTION AND OTHER DOMAIN SPECIFIC NEEDS

The participants expected objects to move on their own, so their behaviors were similar to real-world objects. This is in contrast to the incremental way that animation is accomplished in many systems. This may not come up on all programming tasks and thus might not be considered a language issue. But similar issues can arise in other domains and the usability of the programming system can benefit from analysis of the specific needs of the particular domains in which it will be used. One way may be to provide domain-specific features in the programming language.

6.14. PICTURES

In study one, the experimenter counted how many participants used pictures or diagrams in their solutions, and found that two-thirds of them did. All of these pictures appeared early in the solutions, when setup and layout were being defined. Programming systems should accommodate this form of graphical specification in addition to textual specification.

7. Conclusion and future work

A large part of the programming task is to take a mental plan for solving a problem and transform it into the particular programming language being used. These studies attempt to capture these plans before they undergo the transformation into a programming language. Ideally, the distance between the plans and the programming language would be minimal. However, these studies identify many places where an unnecessarily large gap is imposed by the features and requirements of today's programming languages.

Programming is a task of precision, and one reason that the programming languages may differ from these natural language solutions is that programming languages are more formal and facilitate the expression of solutions with more precision. Indeed, there is a large amount of imprecision and underspecification in the participants' work and it is important to find ways to help beginners to make their specifications more complete. In many cases, however, the structure and algorithms of the natural language solutions are satisfactory, but are in a different style than is allowed in today's programming languages. Future systems should support these approaches, rather than requiring the solutions to be transformed into different, less natural forms.

The new programming language that the authors are designing for children has been influenced significantly by these studies. For example, it will support an event-based style of programming as well as aggregate data access through set creation and manipulation. In order for this latter feature to be effective, it is necessary to improve the accuracy of query specification; these studies show many serious problems with the boolean operators *and*, *or* and *not*. A subsequent study proposes and tests several alternatives to textual boolean expressions, with promising results (Pane & Myers, 2000).

These studies, along with the results of other human-centered research about programming, are resources that can be used to guide and evaluate programming language designs. In addition to the new language for children, the *Natural Programming Project* is using this approach to design several new languages in other domains where it would be useful for non-programmers to have the capabilities of programming. These include authoring of multimedia presentations, programs for inclusion on worldwide web pages, and scripts for automating repetitive tasks in direct manipulation interfaces. Hopefully, this approach will lead to languages that are easier to learn and use than existing languages. The results reported in this article should also be useful to the designers of other new programming languages.

The authors wish to thank the following people for their contributions to this research: Wayne Gray, Albert Corbett, John Chang, Carol Beavers, John Meighan, the participants and analysts, and the anonymous reviewers of an earlier draft of this article.

References

BIERMANN, A. W., BALLARD, B. W. & SIGMON, A. H. (1983). An experimental study of natural language programming. *International Journal of Man-Machine Studies*, **18,** 71–87.

BONAR, J. G. & CUNNINGHAM, R. (1988). Bridge: tutoring the programming process. In J. PSOTKA, L. D. MASSEY & S. A. MUTTER, Eds. *Intelligent Tutoring Systems: Lessons Learned*, pp. 409–434. Hillsdale, NJ: Lawrence Erlbaum Associates.

BONAR, J. & SOLOWAY, E. (1989). Preprogramming knowledge: a major source of misconceptions in novice programmers. In E. SOLOWAY & J. C. SPOHRER, Eds. *Studying the Novice Programmer*, pp. 325–353. Hillsdale, NJ: Lawrence Erlbaum Associates.

BOURNE, L. E. (1966). *Human Conceptual Behaviour*. Boston: Allyn & Bacon.

BRUCKMAN, A. & EDWARDS, E. (1999). Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 207–214. Pittsburgh, PA: ACM Press.

CYPHER, A. & SMITH, D. C. (1995). KidSim: end user programming of simulations. *Proceedings of CHI'95 Conference on Human Factors in Computing Systems*, pp. 27–34. Denver: ACM.

DÉTIENNE, F. (1990). Difficulties in designing with an object-oriented programming language: an empirical study. *Proceedings of INTERACT '90 Conference on Computer-Human Factors*, pp. 971–976. Cambridge, England.

DU BOULAY, B. (1989). Some difficulties of learning to program. In E. SOLOWAY & J. C. SPOHRER, Eds. *Studying the Novice Programmer*, pp. 283–299. Hillsdale, NJ: Lawrence Erlbaum Associates.

GALOTTI, K. M. & GANONG III, W. F. (1985). What non-programmers know about programming: natural language procedure specification. *International Journal of Man–Machine Studies*, **22,** 1–10.

GLASS, R. L. (1955). OO claims — naturalness, seamlessness seem doubtful. *Software Practitioner*.

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

GREEN, T. R. G. & PETRE, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, **7,** 131–174.

GRICE, H. P. (1975). Logic and conversation. In P. COLE & J. MORGAN, Eds. *Syntax and Semantics III*: *Speech Acts*. New York: Academic Press.

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

HOC, J.-M. & NGUYEN-XUAN, A. (1990). Language semantics, mental models and analogy. In J.-M. HOC, T. R. G. GREEN, R. SAMURÇAY & D. J. GILMORE, Eds. *Psychology of Programming*, pp. 139–156. London: Academic Press.

HUTCHINS, E. L., HOLLAN, J. D. & NORMAN, D. A. (1986). *Direct Manipulation Interfaces*. Hillsdale, NJ: Lawrence Erlbaum Associates.

LEWIS, C. & OLSON, G. M. (1987). Can principles of cognition lower he barriers to programming? In G. M. OLSON, S. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers*: *Second Workshop*, pp. 248–263. Norwood, NJ: Ablex.

MILLER, L. A. (1974). Programming ny non-programmers. *International Journal of Man–Machine Studies*, **6,** 237–260.

MILLER, L. A. (1981). Natural language programming: styles, strategies, and contrasts. *IBM Systems Journal*, **20,** 184–215.

MYERS, B. A. (1998). *Natural programming*: *project overview and proposal*. Human–Computer Interaction Institute Technical Report CMU-HCIL-98-100. Pittsburgh, PA: Carnegie Mellon University.

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

NEWELL, A. & CARD, S. K. (1985). The prospects for psychological science in human–computer interaction. *Human–Computer Interaction*, **1,** 209–242.

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

PANE, J. F. & MYERS, B. A. (1996). *Usability issues in the design of novice programming systems*. School of Computer Science Technical Report CMU-CS-96-132. Pittsburgh, PA: Carnegie Mellon University.

PANE, J. F. & MYERS, B. A. (2000). Tabular and textual methods for selecting objects from a group. *Proceedings of VL 2000: IEEE Symposium on Visual Languages*. Seattle, WA (to appear).

PANE, J. F., RATANAMAHATANA, C. A. & MYERS, B. A. (2000). *Analysis of the language and structure in non-programmers' solutions to programming problems*. School of Computer Science Technical Report. Pittsburgh, PA: Carnegie Mellon University.

PEA, R. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, **2**.

SAMMET, J. E. (1981). The early history of COBOL. In R. WEXELBLAT, Ed. *History of Programming Languages*. New York: Academic Press.

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

SOLOWAY, E., BONAR, J. & EHRLICH, K. (1989). Cognitive strategies and looping constructs: an empirical study. In E. SOLOWAY & J. C. SPOHRER, Ed. *Studying the Novice Programmer*, pp. 191–207. Hillsdale, NJ: Lawrence Erlbaum Associates.

SPOHRER, J. G. & SOLOWAY, E. (1986). Analyzing the high frequency bugs in novice programs. In E. SOLOWAY & S. IYENGAR, Eds. *Empirical Studies of Programmers*, pp. 230–251. Washington, DC: Ablex Publishing Corporation.

THIMBLEBY, H., COCKBURN, A. & JONES, S. (1992). HyperCard: an object-oriented disappointment. In P. GRAY & R. TOOK, Eds. *Building Interactive Systems: Architectures and Tools*, pp. 35–55. New York: Springer-Verlag.

WASON, P. C. (1959). The processing of positive and negative information. *Quarterly Journal of Experimental Psychology*, **11**.