To appear in Symposium on Empirical Studies of Programmers (ESP02), Proceedings of 2002 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2002), Arlington, VA, September 3-6, 2002.

Using HCI Techniques to Design a More Usable Programming System

John F. Pane[1], Brad A. Myers, and Leah B. Miller
Computer Science Department and Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
pane+@cs.cmu.edu

Abstract

A programming system is the user interface between the programmer and the computer. Programming is a notoriously difficult activity, and some of this difficulty can be attributed to the user interface as opposed to other factors. Historically, the designs of programming languages and tools have not emphasized usability. This paper describes the process we used to design HANDS, a new programming system for children that focuses on usability, where HCI knowledge, principles, and methods guided all design decisions. The features of HANDS are presented along with their motivations from prior empirical research on programmers and new studies conducted by the authors. HANDS is an event-based language that features a concrete model for computation, provides operators that match the way non-programmers express problem solutions, and includes domain-specific features for the creation of interactive animations and simulations. In user tests, children using HANDS performed significantly better than children using a reduced-feature version of the system where more traditional methods were required to solve tasks.

1. Introduction

Only a very small fraction of users can program their computers. Yet most could benefit in some way from having this capability, whether to customize and interconnect their existing applications or to create new ones. The creators of Boxer [1] said that, similar to writing, "the significance of programming derives not only from the carefully crafted works of a few professionals, but also from the casual jottings of ordinary people." For these ordinary people, understandability, familiarity, ease of performing small tasks, and a usable interface are more important features than technical objectives like formal simplicity, efficiency, verifiability, and uniformity.

Many people who try to learn programming are quickly discouraged because it is a very difficult skill to acquire. Even people who have been trained to program find it to be a challenging task. Why is programming so difficult? Part of the problem is that it requires problem solving skills and great precision. But even people who can envision a viable solution to a programming problem often find it very difficult to express the solution correctly in the form required by the computer. This user-interface problem has long been recognized [2].

The field of HCI has general principles and heuristics that can be applied to help overcome these problems: be consistent, keep it simple, speak the user's language, prevent errors, help the user get started, etc. [3]. The *cognitive dimensions* framework lists additional criteria that can be used to evaluate design alternatives in programming systems, such as closeness of mapping, viscosity, hidden dependencies, imposed guess-ahead and visibility, etc. [4].

Much progress had been made to understand the cognitive demands of programming and the sources of difficulty in existing programming languages and tools. However, this work is generally diagnostic rather than prescriptive, and it has had little influence on the designs of new systems. When language designers face design decisions that are not determined by their technical objectives, they usually choose solutions that are similar to existing languages or that appeal to their intuition. Usability is rarely adopted as a formal objective.

The focus of our work has been to elevate usability to be a first-class objective in programming system design [5]. We have organized the research from Psychology of Programming (PoP), Empirical Studies of Programmers (ESP), and related fields so that it can be readily included among the guidelines and strategies that are used by all language designers [6]. Where we have identified questions that were not yet answered, we have conducted user studies to address them. While our attention has been on beginner programmers, the approach applies equally well to experts, where training and productivity can be impacted.

We propose that all programming language designers should formally include usability principles among the various design guidelines that they use in creating new languages. Depending on the constraints of the project and the target audience, usability may be given more or less weight, but it is always worth considering for those decisions that are not already determined by the other design criteria. For

1. Currently at: RAND, 201 N. Craig St., Pittsburgh, PA 15213 USA

our new programming language for children, we adopted the extreme position of placing usability above all other objectives, while still aiming for a general purpose language. We have explicitly given lower priority to issues such as efficiency, being able to prove correctness, similarity to existing languages, etc. Throughout the creation of this language, our primary guidance for making design decisions came from the HCI, ESP and PoP literature, and new studies of children who do not know how to program.

The target audience for our system is children in fifth grade or older. We selected children because they often have an interest in learning how to program but can be quickly discouraged when they try. Their goals are ambitious – they would like to create programs that are similar to the games and simulations they use every day. These programs are graphically rich and highly interactive, unlike the first programs they are likely to make in many systems, such as to display "hello world" on the screen. Our goal is to provide an easy entry into creating these interactive graphical programs, and to scale well so that it is possible to create more elaborate programs.

Our new programming system for children is called HANDS (Human-centered Advances for the Novice Development of Software). This paper summarizes the HANDS system and describes how its unique set of features were motivated by usability issues [7]. HANDS is a successful case study on how this new design method can yield languages with usability advantages over existing programming systems.

2. Related work

Logo [8] is a successful and popular language for children. Its textual language is based on Lisp, with a syntax that was redesigned to be easier to learn and read, yet it uses unusual punctuation and cryptic names for commands. Logo uses a turtle metaphor for a drawing pen. StarLogo extends the metaphor to parallel processing [9].

Boxer [1] uses a two-dimensional, visible, concrete metaphor where boxes and their spatial relations represent the computation. Variables can be modified by direct manipulation, and the state and code for graphical objects is packaged with their graphical representation. Boxer's textual language is very similar to Logo, with extensions to broaden its range of capabilities.

ToonTalk is a children's programming language based on a video game metaphor [10]. Its cartoon world provides concrete realizations of all of the concepts required in concurrent constraint programming. Programs are constructed by using video-game controls to train robots. However, the low-level primitives of this system present a challenge for beginners, who may have great difficulty in figuring out how to compose the primitives to accomplish their higher-level goals.

AgentSheets [11] is the first of a family of rule-based graphical programming environments that use a spreadsheet metaphor. In these systems, program objects occupy cells in a grid, and interact with the objects in neighboring cells. These interactions are specified by graphical rewrite rules, which are before-and-after pictures. Stagecast (formerly KidSim and Cocoa) extended this metaphor with the capability to use programming by demonstration to create the graphical rewrite rules [12]. While beginners can quickly create some interesting programs, some kinds of games and simulations are difficult to implement. For example, the grid makes it difficult to implement smooth motions in arbitrary directions. Also, graphical rewrite rules can suffer from combinatorial explosion in the number of situations that must be considered; and these rules by their nature are not suited to interactions at a distance.

SmallTalk is an exploratory object-oriented language that was designed to be accessible to non-technical people [13]. A recent portable implementation named Squeak includes a learning environment interface for children, with support for them to add behaviors to objects [14]. This interface features a subset of the SmallTalk language in a more verbose style, with a tile-based structure editor to assist in constructing correct programs.

Numerous "mini-languages" have been created over the years for teaching programming in an environment that is intentionally limited for simplicity [15]. These languages are used for a short time to allow beginners to learn some programming, before they move to more complete programming languages. Usually these languages are very similar to existing languages, so they generally do not break substantial new ground in language design. However, GRAIL stands out among mini-languages because its creators relaxed this constraint, and adopted usability principles and a pedagogical theory to guide its design [16]. User studies showed that students made significantly fewer errors when using GRAIL.

Visual Basic is a popular end-user programming system for non-programmers. It is a textual event-based language with domain-specific support for forms, dialog boxes and tables, which are common in business tasks. The programming language itself is based on the original Basic language, which has many well-known usability problems [6].

Java and Microsoft's C# contain many usability refinements over C and C++. These changes were based on common problems experienced by programmers using the earlier languages. However the designers were constrained in how far they could deviate from the predecessors, so programmers could more easily switch to the new languages.

3. User studies of non-programmers

One goal of the new system is to provide a *close mapping* between the way the programmer envisions a problem solution and the expression of that solution in program code [4]. We conducted a pair of studies to examine the language and structure that children and adults use before they have been exposed to programming [17]. In these studies, we presented programming tasks to non-programmers and they solved them on paper. The tasks covered a broad set of essential programming techniques and concepts, such as control structures, storage and manipulation of data, arith-

metic, Boolean logic, searching and sorting, animation, interactions among objects, etc. In posing the problems, we used pictures and very terse captions, to minimize the chance that our materials would influence the answers.

Some of the features observed in these studies were: an event-based or rule-based structure was used, where actions were taken in response to events; aggregate operators (acting on a set of objects all at once) were used much more often than iteration through a set and acting on the objects individually; data structures were avoided by using content-based queries; a natural language style was used for arithmetic; objects automatically remembered their state (such as motion), and the participants only mentioned changes in this state; operations were more consistent with list data structures than arrays; and participants rarely used Boolean expressions, but when they did their expressions were often incorrect if interpreted according to the rules of most programming languages. All of these observations have influenced the design of our new programming system.

4. The HANDS system

This section describes the various usability issues we considered, and how the HANDS system was designed to address them.

4.1. The role of the programming environment

The programming environment comprises the tools for viewing, editing, debugging and running the program. Many programming languages are designed independently from the environment, but the HANDS language was not. We designed the environment to work in tandem with the language to address some important usability issues. For example, editing support can relieve many of the problems that people have with writing programs in a textual language. The environment's components play critical roles in the overall usability of the HANDS system.

4.2. Textual vs. visual

One of the first decisions we faced was whether the language should be textual or visual. In visual languages, graphics replace some or all of the text in specifying programs. Proponents of visual programming languages often argue that reducing or eliminating the text in programming will improve usability [12]. However, much of the underlying rationale for this expectation is suspect [18]. User studies have shown mixed results on the superiority of visual languages over text (e.g. [19]), and the advantage of visual languages tends to diminish on larger tasks.

The participants in our user studies usually drew pictures describing the layout of the program, and then used text to describe the behaviors and actions. HANDS supports this hybrid approach, and relies on the programming environment to alleviate some of the difficulties of textual languages. During program entry, context-sensitive menus make it easier to know what choices are available and to enter the program correctly. This support could be augmented with a tile-based editor, as seen in Squeak [14] and other systems. Also, the system could provide visual representations for textual elements that are difficult. One example is Boolean expressions, as discussed below.

4.3. Language syntax

The programming language should observe the HCI principle to speak the user's language. This means it should avoid using words and symbols that are unfamiliar or that have different meanings in other domains (e.g. void, static, dim, etc.). If users are not sure what to do they often use their knowledge of other areas including natural language and mathematics [20]. If the syntax or semantics of the programming language are incompatible with this existing knowledge, errors and confusion result. For example, **x = x + 10** does not make sense in mathematics.

The syntax of HANDS was designed to match the common ways that participants expressed operations in our studies. For example, the system accepts a natural-language style for arithmetic: (e.g. **add 100 to score**). Overall, the language has a verbose conversational style, similar to HyperTalk. To improve readability, the system allows the word **the** to be placed anywhere in the code – it is ignored.

Many languages use a common symbol (such as end) to terminate many different kinds of control structures. When several structures are nested, it can be difficult to figure out which terminator belongs to which structure. If the terminator is optional, additional ambiguities can result. For example, Pascal and HyperTalk have the dangling-else problem, where the system may attach an else clause to a different if statement than the user intended. For these reasons, all structures in HANDS that allow statements to be nested require a matching terminator that incorporates the name of the structure. For example, the **if** statement is terminated with **end if**. To reduce the amount of typing required, the system automatically inserts these terminators.

Where possible, we avoided requiring punctuation or other symbols if their only purpose was to make it easier to parse. These syntactic elements are a distraction from the semantically-important parts of the program, and are a common source of errors. For example, no semicolon is required to terminate or separate statements, and parenthesis are only required when expressions are being nested and interpretation would otherwise be ambiguous.

4.4. Representation of the program

The von Neumann computational model is an obstacle for beginners because it is unfamiliar and has no real-world counterpart [21]. Beginners must learn, for example, that the program follows special rules of control flow for procedure calls and returns. Usability could be improved by providing a different model for the computation that is concrete and familiar [12].

Most languages also have 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 [22]. The programming system should make information visible or readily accessible any time it is relevant.

The HANDS system addresses these problems with a new model of computation that is concrete, uses familiar concepts, and has high visibility. In HANDS, an agent named Handy sits at a table, manipulating information on cards (see Figure 1). All of the data in the system is stored on these cards, which are global, persistent and visible on the table. Each card must have a unique name, which is not case sensitive. There are no local variables, although a temporary naming mechanism is available (see Section 4.12).

The front of each card contains a list of name-value pairs called properties. Several properties are always present: the **cardname** property holds the card's name, and the **x** and **y** properties contain the card's position coordinates. The programmer can add more properties as needed, so cards are similar to records (or structs) in other languages. Each property on a card has a unique name, which is not case-sensitive. The programmer refers to the nectar property of a card named flower with one of these syntaxes: **nectar of flower** or **flower's nectar**. In addition to the usual ways that

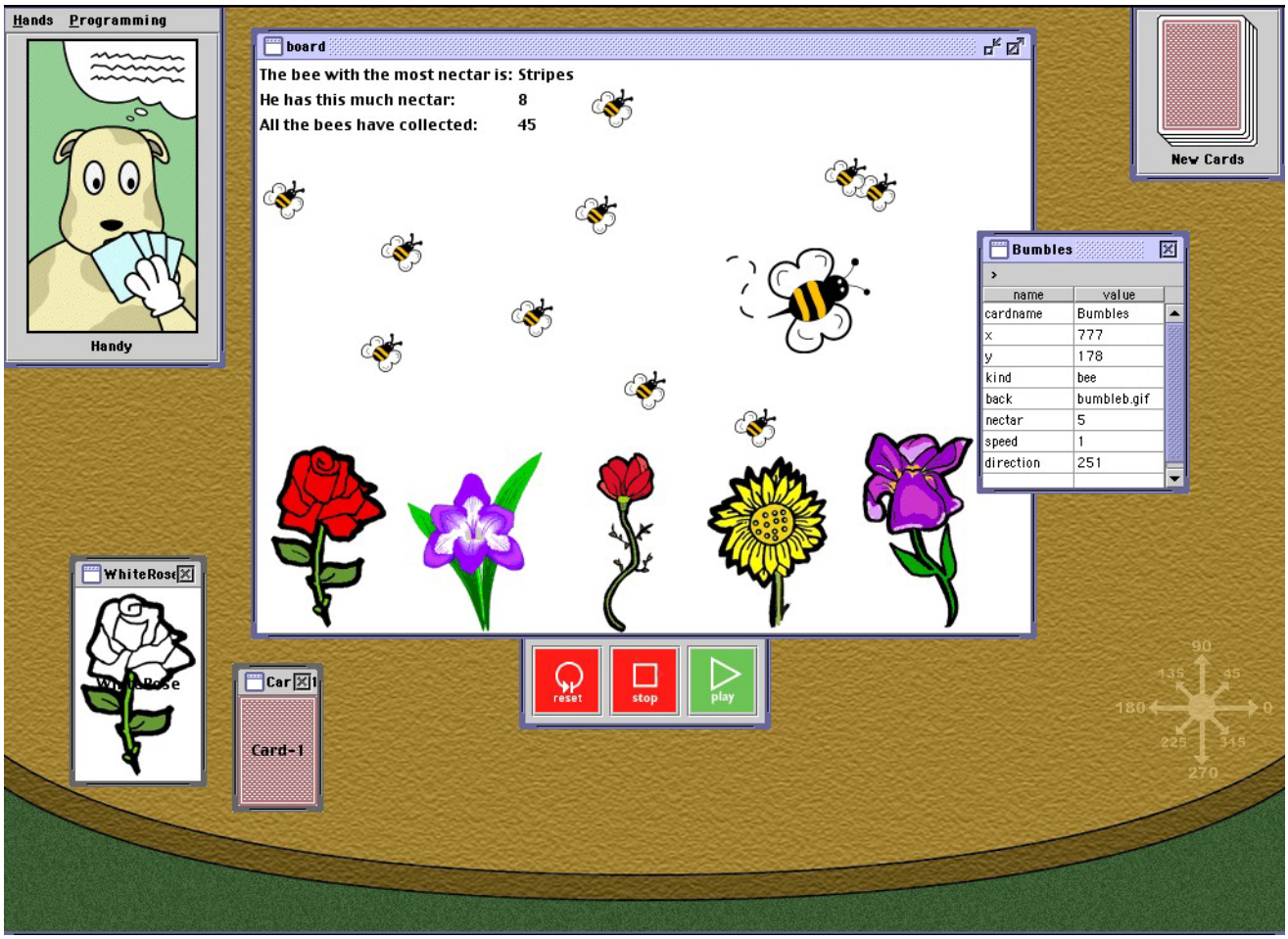

Figure 1. The HANDS system portrays the components of a program on a round table. All data is stored on cards, which can be drawn from the pile at the top right and dragged into position. At the lower left, two cards are shown face-down on the table. One has a generic card back and the other has been given a picture by the programmer. In the center of the table is a board, where the cards are displayed in a special way where only the contents of the back are displayed. Each picture, string, and number on the board is a card. At the right, one of the cards has been flipped face-up, where its properties can be viewed and edited. The programmer inserts code into Handy's thought bubble, by clicking on Handy's picture in 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. The stop button halts the program, and the reset button will restore all cards to their state at the time the play button was last pressed. For reference, a compass is embossed on the table at the lower right.

data can be manipulated by the code of a running program, cards and their properties can be managed by direct manipulation, even before or after the program has run.

Some properties are treated specially by the system. For example, the **back** property, if present, controls what is displayed on the back of the card when the card is face-down. If the value of the **back** property is the name of a file containing an image, the image is displayed on the back of the card. Otherwise, the value in the property is displayed literally on the back of the card. If there is no **back** property on a card, a generic card back is displayed when it is face-down. All cards are flipped face-down when the program starts running. These features make it very easy for the programmer to display graphics and text on the screen. Figure 1 shows examples of cards with pictures, strings and numbers on their backs, as well as one card with the generic picture on its back.

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.

A future version of the HANDS system will permit the programmer to generate an end-user version of any program, giving it a more completed look. In this version, most of the HANDS environment will be invisible, and the computer will only show the board, the large white region at the center of Figure 1. Cards that are on the board are therefore the only ones that are visible in the end-user version. Other cards used in the computation can be placed elsewhere on the table.

4.5. Programming style and model of execution

HANDS is event-based, to match the style of programming that we observed 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 one or more imperative statements to execute in response to the event. After these statements have executed, control returns to the system, where the next event is dispatched. Event handlers can be triggered by the following kinds of events: the program starting to run or stopping; an object appearing, disappearing, or changing; objects colliding; objects being clicked by the user; keystrokes; nothing happening (the idle loop); or something happening (any event). If an event is generated and there is no handler for it, the system continues on to the next event in the queue. Figure 2 shows the browser for event handlers.

4.6. Data types

The value of a property can hold any of the following types of data: an identifier, representing the name of a card; a string literal, delimited by quotes; a numeric literal, either integer or floating point; a Boolean literal, either yes or no; a list of zero or more data elements, not necessarily of homogeneous type.

The system does not enforce types until necessary. An error will be reported only if the system cannot interpret a data value as the expected type, such as if a math operation is performed on a non-numeric value. All data types can successfully be interpreted as strings. Lists have properties like traditional lists, such as being unbounded and permitting insertion without making space, and properties like arrays, such as index-based access.

4.7. Operations on Cards

The programmer can instruct Handy to create and delete cards, and to modify the properties of cards. If the property does not exist, it is created.

- **set nectar of flower to 100**
- **set flower's nectar to flower's nectar + 25**
- **add 25 to flower's nectar**

The last two examples accomplish the same effect; the latter uses the more natural syntax observed in our studies

4.8. Aggregate operations

Several language constructs or features have been repeatedly identified as troublesome in the literature. For example, researchers have identified ways to improve performance by redesigning loops [23], yet these solutions have not been adopted by most languages. More importantly, many languages force users to perform iteration in situations where aggregate operations could accomplish the task more easily [24].

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. Most languages force the programmer to use iteration, violating the principle of closeness of mapping.

HANDS has full support for aggregate operations. All

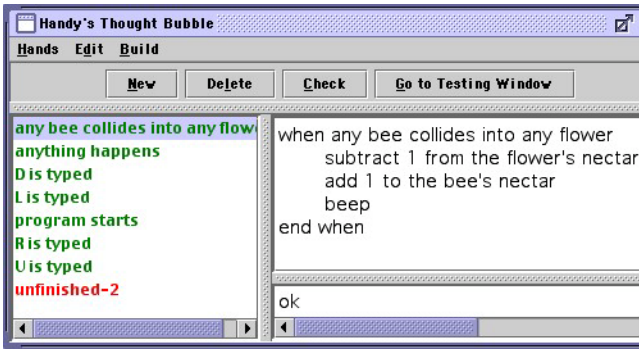

Figure 2. HANDS is an event-based system. The left pane lists seven complete (syntactically correct) event handlers, and one that is marked in red because it is not finished (unfinished-2). The upper right pane shows the code for *when any bee collides into any flower*. The lower right pane displays any error messages for the selected event handler.

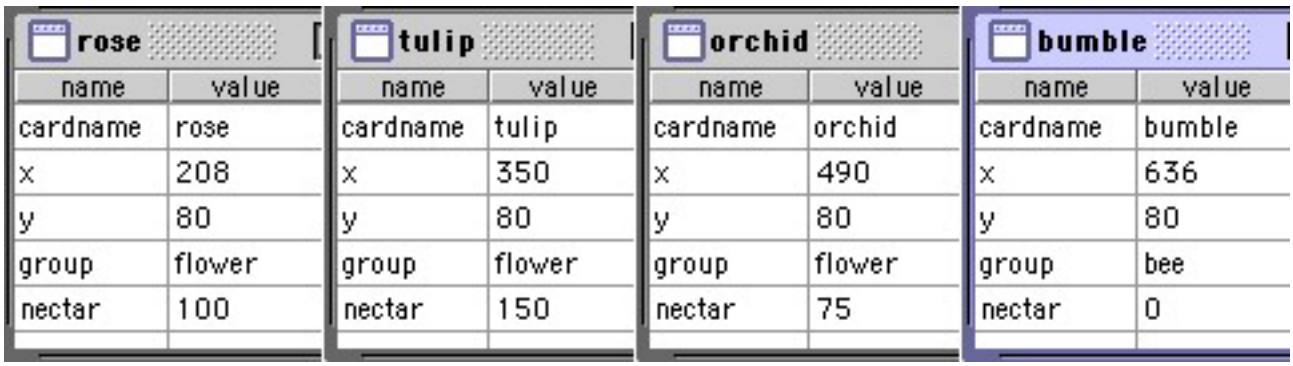

Figure 3. When the system evaluates the query *all flowers* it returns *orchid, rose, tulip*.

operators can accept lists as well as singletons as operands, or even one of each. For example,

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

4.9. 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. 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. If the value is a word ending in "s", it will also match cards that have the value without the trailing s. Figure 3 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**
- **all (flower and (nectar < 100))** evaluates to **orchid**

Section 4.10 describes a more effective method for specifying more complex queries like the last example.

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

- **set the nectar of all flowers to 0**

4.10. Boolean expressions

The accurate specification of Boolean expressions is an area that is very difficult and has been studied extensively. The common uses of the words AND and OR in natural language lead to errors when these words are used to name the Boolean operators in queries; and the intended scope of the NOT operator is ambiguous.

To address this problem, we examined using different keywords and structures to specify Boolean expressions, as well as a tabular alternative called match forms that we designed to be similar to cards (Figure 4). On a match form, all of the listed values implicitly form a conjunction. Negation is specified by prefacing a value with the NOT operator. Disjunction is specified by including an additional match form adjacent to the first one. The match forms in Figure 4 represent the query **(blue and not square) or (circle and not green)**.

In our user study, match forms performed better than any of the purely textual representations we tested, including Boolean expressions [25]. Match forms helped the participants avoid some of the common problems they had with using the Boolean operators to construct textual expressions. These errors included inconsistently using AND for both conjunction and disjunction, and inconsistently applying precedence to operators, especially when the NOT operator was present.

Match forms will be incorporated into the HANDS system at a future date, with some extensions. Although match forms can express arbitrarily complex queries in disjunctive normal form, this is sometimes less concise than unrestricted Boolean expressions would allow. This will be relieved somewhat by allowing an entire form to be negated ("objects that do not match ..."). The match forms in HANDS will also have property names alongside the values, like cards do, so the programmer can easily restrict the match of a value to a specific property.

4.11. List operators

The system provides a basic set of list operators, as seen in Lisp and other languages. Like all operators in HANDS, these operators also accept empty lists and singletons. Here

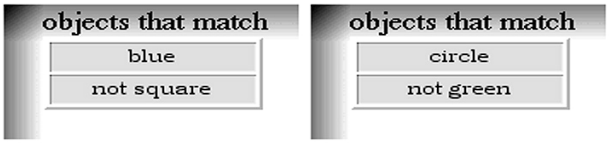

Figure 4. Match forms expressing the query *(blue and not square) or (circle and not green)*.

are some examples, referring again to Figure 3:

- **FirstItem of all flowers** evaluates to **orchid**
- **GreatestItem in nectar of all flowers** evaluates to **150**
- **Sorted nectar of all flowers** evaluates to **75,100,150**
- **CardWithLeast nectar of all flowers** evaluates to **orchid**
- **Sum nectar of all flowers** evaluates to **325**

It is interesting to compare the last example (Sum) with how it might look in a typical programming language:

```
int sum = 0;
for (i=0; i<cards.length(); i++) {
    if (cards[i].containsValue("flower")) {
        sum += cards[i].nectar;
    }
}
return sum;
```

This solution requires temporary variables, three kinds of parenthesis or brackets, three other kinds of punctuation, and the complexities of iteration and array indexing.

4.12. Loop and conditional control structures

The aggregate and list operators greatly reduce the need for iteration in HANDS. However, one high-level loop control structure is available, if needed. This example sets the nectar properties of the rose, tulip and orchid cards to a different random value between 50 and 100:

```
with all flowers calling each the flower
    set the flower's nectar to random from 50 to 100
end with
```

The clause **calling each the flower** is a temporary naming mechanism that causes the identifier flower to be bound to one of flower during each iteration of the loop. If the user does not explicitly set this name, the system uses the identifier **item**.

```
with all flowers
    set the item's nectar to random from 50 to 100
end with
```

A general if statement incorporates the functionality of case and cond statements in other languages:

```
• if f=tulip then                          // if-then-else style
    set f's nectar to 0
    otherwise set f's nectar to 100
end if
• if f=…                                   // case statement style
        tulip then set f's nectar to 0
        orchid then set f's nectar to 10
        otherwise set f's nectar to 20
end if
• if                                       // cond style
        f's nectar > 30 then set f's nectar to 0
        f's nectar < 30 then set f's nectar to 10
        otherwise set f's nectar to 20
end if
```

We observed each of these notations in our studies, and there is no good reason that to distinguish them as three different control structures with inconsistent names and syntaxes. When there are multiple conditions, only the first one to match is executed. The otherwise clause is optional.

4.13. 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 generates events for collisions among objects and for keyboard and mouse input from the user.

It is easy to create graphical objects and text on the screen, as described above in Section 4.4. Any card that contains properties named speed and direction is automatically animated by the system without any programming. Speed is a relative value, and can be positive or negative. Direction is an angle specified in degrees, adopting the convention from math that zero points to the right and the angle increases in a counter-clockwise direction. Since some users may not be familiar with this convention, an image of a compass is available in HANDS for the user to refer to when working with directions (see Figure 1).

This combination of features permits the programmer to implement sophisticated behaviors with only a few lines of code. For example, after giving the bees in Figure 1 initial speeds and directions, the programmer can use this event handler to make them fly around like bees:

```
when any bee changes
    add random from -5 to 5 to the bee's direction
end when
```

Each time the system moves one of the bees, an event is generated indicating that the card has changed. This event handler responds to that change by making a small random change to the bee's direction. A future version of HANDS will also provide timers that generate events at specific times or intervals.

4.14. Modularity and encapsulation

HANDS programs can be extended by importing one or more existing programs. The system integrates the new program by adding the cards to the table and adding the event handlers to the thought bubble. If a handler exists for a particular event in both programs, the system offers to merge the code automatically. This makes it very convenient to build and use a library of small autonomous objects, each as a small program with one card and the code to control its behavior. For example, the bees from the program shown in Figure 1 could be imported into another program. The bees would fly around in the new program, and if there are flowers present the bees would try to collect nectar from them. But if there were no flowers, the bees' nectar-collecting code would not interfere with the operation of the program.

The HANDS design can support additional agents besides Handy, working together or in competition to accomplish tasks. This extension includes privacy of data because cards can be held in one agent's hand making them invisible to other agents. This capability is not visible in the current user interface because we determined it is too confusing for beginners. However, as the program grows larger and the programmer becomes more experienced, this

encapsulation feature becomes quite useful. For example, it would permit imported programs to be kept separate from the existing code.

We have a design for extending the card metaphor to represent procedural abstraction. Parameters will be passed by setting the properties of one of these cards. Then, when the result property is read, the procedure would be called to compute the value of the property. In addition, we would like to investigate how the system scales to larger programs. For example, we will look at ways to help the programmer manage large quantities of cards and event handlers.

5. User study

Our user-centered design process resulted in programming system that has a unique combination of features. Many of these features are not present in currently-popular programming systems even though they are not new ideas. We selected three such features for evaluation, to investigate whether their inclusion has an impact on usability [26].

The three features examined in the study are: *queries*, *aggregate operations*, and the *high visibility of data*. In other popular programming systems such as Logo or Java, programmers use data structures, iteration, and debuggers to accomplish the tasks that might otherwise be accomplished with these three HANDS features. It happens that HANDS also supports these alternative methods.

A version of HANDS was constructed where the three studied features were turned off. With this limited version, programmers could achieve all of the tasks we gave them, but had to use the authentic alternative methods that are used in other programming systems. The study compared the HANDS system with this limited version.

Twenty three fifth-grade children from a public school in Pittsburgh, PA, USA volunteered to participate in this study. None of them had any programming experience. Half of them used the HANDS system and the other half used the limited version. In either case, the participants spent three hours in the study, first learning the basics of the system from a tutorial and then trying to accomplish a series of tasks. In working through the tutorial, the participants began with an empty program and created several flowers and a bee that flies around collecting nectar from the flowers. The bee was controlled by keyboard commands, and the program displayed some basic statistics about the amount of nectar the flowers had and which flower had the least nectar. After completing the tutorial, the participants loaded a program similar to the one shown in Figure 1, which had more bees and some pre-defined cards to help solve the tasks. They used the remaining time in the session to work on six tasks that required them to add code to this program.

All of these materials were constructed to be as similar as possible in the two versions, differing only where necessary due to the limitations of the reduced-feature version of HANDS. The tutorial for the limited system was derived from the full-featured tutorial. Those portions utilizing a feature that was missing in the limited system were replaced with material teaching the easiest way to use the system's remaining features to achieve the same result. This modification resulted in a minor increase in the size of the tutorial.

The study measured performance on the six tasks. Of the children who finished the tutorial and began working on the tasks (9 in each group), participants using HANDS performed significantly better than their peers using the limited system ($p<.05$). Overall, the children using HANDS correctly solved 19 tasks, and seven of them solved at least one task. Only one of the participants using the limited system was able to solve any tasks, and this child only solved one. These differences cannot be attributed to extra time spent on the slightly longer tutorial in the limited condition.

The study demonstrates the combined impact of the three features – queries, aggregate operations, and data visibility – but of course does not tease apart their individual contributions. Nonetheless, it validates the design process that led to the features' inclusion in HANDS, and suggests that these features could improve the usability of programming systems in general. More work is planned to assess the overall effectiveness of HANDS and compare it with other programming systems for children such as Logo, Boxer or Stagecast.

6. Range and Scalability

HANDS was designed to be easy to learn, but an additional goal was for HANDS to be powerful and general, so that more experienced programmers will not quickly outgrow the system. To explore the range and scalability of HANDS, several more experienced programmers have implemented a broad set of programs. One of the authors created a program to compute prime numbers with a single 8-line event handler and six cards. His solution to the *Towers of Hanoi* problem required six event handlers (53 lines of code) and ten cards.

An undergraduate computer science student used the system to build a version of the game *Breakout*. A two-level version of this game required 12 event handlers (178 lines of code), and 62 cards. 53 of the cards represent bricks. Each additional level added to the game would require about 25 more cards and 15 more lines of code.

This student also implemented a simulation of the ideal gas law, where the relationships among pressure, volume, and temperature obey the formula: PV=nRT. This program displays a chamber with small molecules bouncing around inside. The user can manipulate the variables, and observe the effect on the other variables and see changes in the speeds of the molecules. This simulation required 18 rules (180 lines of code), and 36 cards. 12 rules and 12 cards implemented checkboxes and scrollbars. A future version of HANDS may provide a library of widgets, reducing the size of this simulation accordingly.

A high school student compared HANDS with Stagecast [12] for a science project. He implemented Pacman in both systems, and concluded that HANDS was easier to use, required fewer lines of code, and enabled him to implement

more features.

7. Conclusions

This paper is a case study of a new, human-centered approach to the design of programming languages. It tracks the design of a new programming system for children, describing how HCI techniques and evidence from the literature, as well as new studies by the authors investigating unaddressed questions, impacted the design and selection of features for the language.

This human-centered design approach has led to a new programming system with a set of features that differs substantially from the currently-popular programming languages. Several of these features have already been shown to have a significant positive effect on usability. It is a promising validation of the design methodology described here, and suggests that this method would be generally useful for all programming language designers.

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

References

[1] A. A. diSessa and H. Abelson, "Boxer: A Reconstructible Computational Medium," *Communications of the ACM*, vol. 29, pp. 859-868, 1986.

[2] A. Newell and S. K. Card, "The Prospects for Psychological Science in Human-Computer Interaction," *Human-Computer Interaction*, vol. 1, pp. 209-242, 1985.

[3] J. Nielsen, "Heuristic Evaluation," in *Usability Inspection Methods*, J. Nielsen and R. L. Mack, Eds. New York: John Wiley & Sons, 1994, pp. 25-62.

[4] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131-174, 1996.

[5] B. A. Myers, "Natural Programming: Project Overview and Proposal," Carnegie Mellon University, Pittsburgh, PA, Human-Computer Interaction Institute Technical Report CMU-HCII-98-100, January 1998.

[6] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems," Carnegie Mellon University, Pittsburgh, PA, School of Computer Science Technical Report CMU-CS-96-132, August 1996.

[7] J. F. Pane, "A Programming System for Children that is Designed for Usability," Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 2002.

[8] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

[9] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. Boston: The MIT Press, 1994.

[10] K. Kahn, "ToonTalk: An Animated Programming Environment for Children," *Journal of Visual Languages and Computing*, vol. 7, pp. 197-217, 1996.

[11] A. Repenning and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *Computer*, vol. 28, pp. 17-25, 1995.

[12] D. C. Smith, A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, vol. 37, pp. 54-67, 1994.

[13] D. H. H. Ingalls, "Design Principles Behind Smalltalk," in *BYTE Magazine*, 1981.

[14] J. Steinmetz, "Computers and Squeak as Environments for Learning," in *Squeak: Open Personal Computing and Multimedia*, M. Guzdial and K. Rose, Eds.: Prentice Hall, 2001, pp. 453-482.

[15] P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller, "Mini-languages: A Way to Learn Programming Principles," *Education and Information Technologies*, vol. 2, pp. 65-83, 1997.

[16] L. K. McIver, "Syntactic and Semantic Issues in Introductory Programming Education," Ph.D. Thesis, School of Computer Science and Software Engineering, Monash University, Australia, 2001.

[17] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, "Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems," *International Journal of Human-Computer Studies*, vol. 54, pp. 237-264, 2001.

[18] A. F. Blackwell, "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, 1996, pp. 240-246.

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

[20] J.-M. Hoc and A. Nguyen-Xuan, "Language Semantics, Mental Models and Analogy," in *Psychology of Programming, Computers and People Series*, J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, Eds. London: Academic Press, 1990, pp. 139-156.

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

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

[23] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," in *Studying the Novice Programmer*, E. Soloway and J. C. Spohrer, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, 1989, pp. 191-207.

[24] L. A. Miller, "Natural Language Programming: Styles, Strategies, and Contrasts," *IBM Systems Journal*, vol. 20, pp. 184-215, 1981.

[25] J. F. Pane and B. A. Myers, "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, 2000, pp. 157-164.

[26] J. F. Pane and B. A. Myers, "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, 2002, pp. 684-685.