An environment that works the way nonprogrammers expect is more inviting and helps users become more confident and productive.

NATURAL PROGRAMMING LANGUAGES AND ENVIRONMENTS

By Brad A. Myers, John F. Pane *and* Andy Ko

OVER THE LAST SIX YEARS, WE HAVE BEEN WORKING TO CREATE PROGRAMMING languages and environments that are more natural, or closer to the way people think about their tasks. Our goal is to make it possible for people to express their ideas in the same way they think about them. To achieve this, we have performed various studies about how people think about programming tasks, both when trying to create a new program and when trying to find and fix bugs in existing programs. We then use this knowledge to develop new tools for programming and debugging. Our user studies have shown the resulting systems provide significant benefits to users.

It is somewhat surprising that in spite of over 30 years of research in the areas of empirical studies of programmers (ESP) and human-computer interaction (HCI), the designs of new programming languages and debugging tools have generally not taken advantage of what has been discovered. For example, the C#, JavaScript, and Java languages use the same mechanisms for looping, conditionals, and assignments shown to cause many errors for both beginning and expert programmers in the C language. Systems such as MacroMedia's Director and Flash, Microsoft's Visual Basic, and general-purpose programming environments like MetroWerks' CodeWarrior and Microsoft's Visual C++, all provide the same debugging techniques available for 60 years: breakpoints, print statements, and showing the values of variables.

Our thorough investigation of the ESP and HCI literature revealed many results that can be used to guide the design of new programming systems, many of which have not be utilized in previous designs. However, there are many significant gaps in our knowledge about how people reason about programs and programming. For example, there has been very little study about which fundamental paradigms of computing are the most natural or what questions people ask when debugging. We are performing user studies which investigate these questions.

It is in this context that we developed the Natural Programming design process, that treats usability as a first-class objective by following these steps:

- Identify the target audience and the domain, that is, the group of people who will be using the system and the kinds of problems they will be working on.
- Understand the target audience, by studying the actual language, techniques, and thinking they naturally use when trying to solve problems. This includes an awareness of general HCI principles as well as prior work in the psychology of programming and empirical studies. When issues or questions arise that are not answered by the prior work, conduct new user studies to examine them.
- Design the new system based on this information.
- Evaluate the system to measure its success, and to understand any new problems the users have. Redesign the system based on this evaluation, and then reevaluate it, following the standard HCI principle of iterative design.

Thus, the Natural Programming approach is an application of the standard user-centered design process to the specific domain of programming languages and environments.

This article provides an overview of some of the work of the Natural Programming project.[1] We explore why naturalness might be better for developers and what might be more natural in programs for graphics and data processing based on initial user studies. The results were used in the design of a new language and environment called Human-centered Advances for the Novice Development of Software (HANDS). We also discuss our survey of programmers creating small and medium-size programs using a different environment called Alice (see www.alice.org).

Why Natural Might Be Better

The premise of our research project is that programmers will have an easier job if their programming tasks are made more natural. By "natural," we mean "faithfully representing nature or life," which here implies it works in the way people expect. By "natural programming" we are aiming for the language and environment to work the way that nonprogrammers expect.

Why would this make programming easier? One way to define programming is the process of transforming a mental plan in familiar terms into one compatible with the computer [3]. The closer the language is to the programmer's original plan, the easier this refinement process will be. This is closely related to the concept of directness that, as part of *direct manipulation*, is a key principle in making user interfaces (UI) easier to use. UI designers and researchers have been promoting directness at least since Ben Shneiderman identified the concept in 1983, but it has not even been a consideration in most programming language designs.

Conventional programming languages require the programmer to make tremendous transformations from the intended tasks to the code design. For example, a typical program to add a set of numbers in C uses three kinds of parentheses and three kinds of assignment operators in five lines of code, whereas a single "SUM" operator is sufficient in a spreadsheet [2]. We argue that if the computer language were to enable people to express algorithms and data more like their natural expressions, the transformation effort would be reduced.

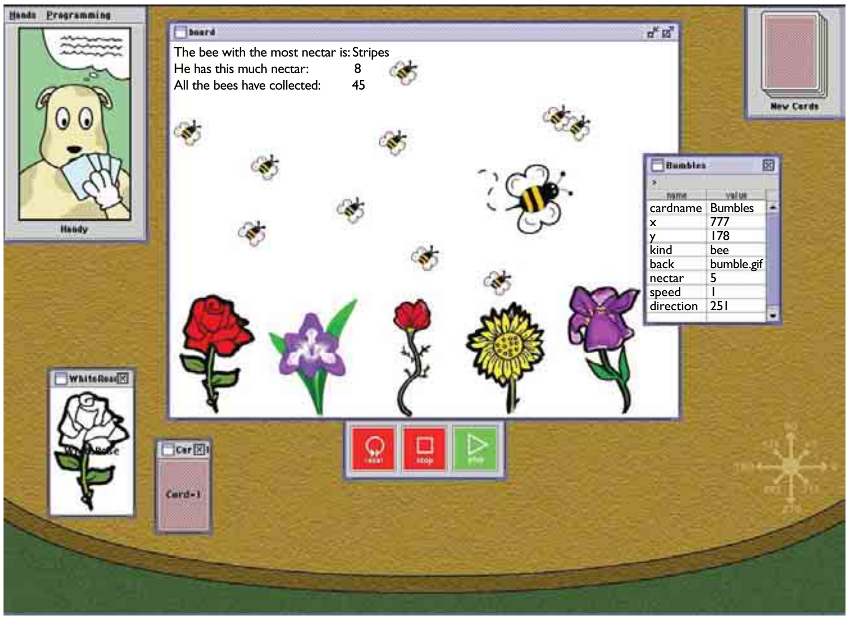
Similarly, debugging activities could benefit from being more natural. Research describes debugging as an exploratory activity aimed at investigating a program's behavior, involving several distinct and interleaving activities [12]:

- Hypothesizing what runtime actions caused failure;
- Observing data about a program's runtime state;
- Restructuring data into different representations;
- Exploring restructured runtime data;
- Diagnosing what code caused faulty runtime actions; and
- Repairing erroneous code to prevent such actions.

Current debugging tools support some of these activities, while hindering others. For example, breakpoints and code-stepping support observation of control flow but hinder exploration and restructuring, whereas visualization tools help restructure data but hinder diagnosis and observation [5]. Yet none of these tools support hypothesizing activities. The argument behind our Natural Programming approach to debugging is that support for such question-related activities will significantly improve success. If programmers have a weak hypothesis about the cause of a failure, any implicit assumptions about what did or

[1]More details are available from other papers at www.cs.cmu.edu/~NatProg).

The goal is to make it possible for people to express their ideas in the same way they think about them.



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

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

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

did not happen at runtime will go unchecked. Not only do these unchecked assumptions cause debugging to take more time [12], but they also result in new errors. For example, in a study of Alice users we found that 50% of all errors were due to programmers' false assumptions in the hypotheses they formed while debugging existing errors [6].

Related work. We build on the research of many others, who have studied programming and debugging. Prior studies (see [10] for a summary) have shown that features of the programming environment are a crucial part of making a programming language effective and easy to use. For example, providing immediate feedback about problems and testing helps with problem solving. The ability to test partial solutions is an important feature for novices and experts alike, especially during testing and when reusing code. This incremental and frequent testing should be encouraged because it is a useful debugging strategy. When novices test their code incrementally, they perform better [2].

The idea that the programming environment can help users construct programs has a long history. "Syntax Directed Editors" (also called "Structure Editors") have been used to help eliminate syntax errors at least since the Cornell Program Synthesizer [11], and there have been many variations. Modern tools like Visual Basic provide context-dependent pop-up menus that insert correctly formatted code, but do not restrict what users can type. Alice takes an extreme stance and only allows syntactically correct statements to be entered, since all editing is performed by dragging-and-dropping statements and using pop-up menus to specify parameters. A number of environments have adapted the successful spreadsheet style of end-user programming to other domains (for example, [1, 4]).

Many tools have been created to help with debugging, but most focus on visualization of data and control flow. Research by Kehoe, Stasko, and Taylor suggests, however, that these are not particularly helpful for debugging [5]. *Communications* devoted a special section to new ideas for debugging in April 1997, but none of the systems have been user tested or widely deployed [8].

Language Studies

We conducted two studies to examine the language and structure that children and adults naturally use in solving problems before they have been exposed to programming. Participants were presented with programming tasks and asked to solve them on paper using whatever text or diagrams they wanted to use. To avoid biasing the subjects' answers, the study materials were constructed with great care, using graphical depictions and terse descriptions of problem scenarios. One study used the PacMan video game and another used database access scenarios more typical of business programming tasks.

Some observations from these studies were:

- An event-based or rule-based structure was often used, where actions were taken in response to events. For example, "When PacMan loses all his lives, it's game over."
- Aggregate operators (acting on a set of objects all at once) were used much more often than iterating through the set and acting on the objects individually. For example, "Move everyone below the 5th place down by one."
- Participants rarely used Boolean expressions, but when they did they were likely to make errors. That is, their expressions were not correct if interpreted according to the rules of Boolean logic in most programming languages.
- Participants often drew pictures to sketch out the layout of the program, but resorted to text to describe actions and behaviors.

Additional details about these studies are reported in [9].

The HANDS Environment and Language

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

HANDS uses an event-based language that features a new model for computation, provides queries and aggregate operators that match the way nonprogrammers express problem solutions, has high visibility of program data, and includes domain-specific features for the creation of interactive animations and simulations.

In HANDS, the computation is represented as an agent named Handy, sitting at a table manipulating a set of cards (see Figure 1). All the data in the system is stored on these cards, which are global, persistent, and visible on the table.

HANDS is event-based—a programming style that most closely matches the problem solutions in our studies. It has full support for aggregate operations—all operators can accept lists or singletons as operands. Lists can be generated as needed, by using query operators that search all of the cards for the ones matching the programmer's criteria. Queries and aggregate operations work in tandem to enable the programmer to concisely express actions that would require iteration in most languages. For example, the following is a typical HANDS statement combining queries and aggregates:

```
set the nectar of all flowers to 0
```

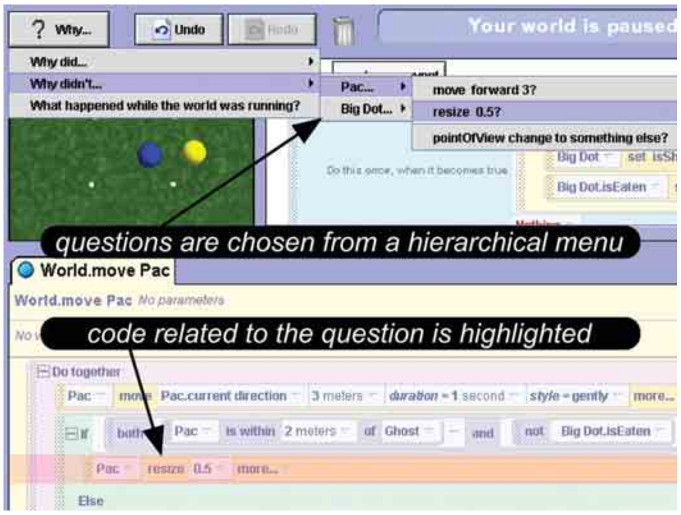

? Why...
Undo
Redo
Your world is paused
Why did...
Why didn't...
What happened while the world was running?
Pac...
Big Dot...
move forward 3?
resize 0.5?
pointOfView change to something else?
questions are chosen from a hierarchical menu
World.move Pac
code related to the question is highlighted
Do together
Pac move Pac.current direction 3 meters duration = 1 second style = gently more...
If both Pac is within 2 meters of Ghost and not Big Dot.isEaten
Pac resize 0.5 more...
Else

Figure 2. The Alice programming environment with a "why didn't" question in progress.

To examine the effectiveness of HANDS, we conducted a user study. Ten-year-olds were able to learn the HANDS system during a three-hour session, and then use it to solve programming problems. Children using the full-featured version of HANDS performed significantly better than their peers who used a version modified to be more like typical programming systems. Additional details about this study are reported in [9].

Debugging Studies

Our studies of the language and structure that people use helped us to design a more natural programming language. We are also performing studies of how programming environments can help programmers avoid making errors, and help them find and fix the errors they have already made. An important part of these studies is determining what causes programmers to make errors in the first place.

We have integrated many strands of prior research on human error, and have found that errors are ultimately caused by long chains of *breakdowns* that happen for one of two reasons: some breakdowns occur in the programmer's head, such as using an inappropriate strategy, or having a misunderstanding about a particular aspect of a programming language. The other type of breakdown is caused by things outside of the programmers' head, in the programming language and environment. For example, when it is difficult to inspect the values of variables at runtime, programmers may have breakdowns in debugging. Or, when a language supports different meanings for the same text (such as the "+" operator in Java), programmers may accidentally introduce errors.

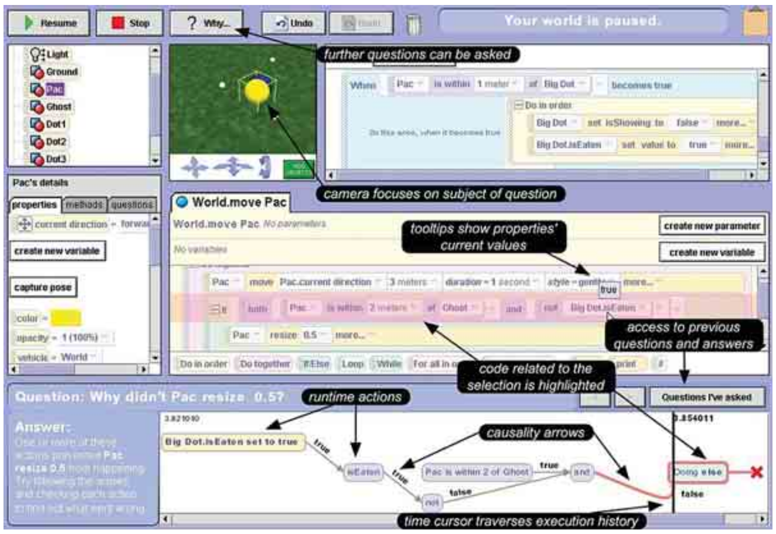

further questions can be asked
camera focuses on subject of question
tooltips show properties' current values
access to previous questions and answers
code related to the selection is highlighted
runtime actions
causality arrows
time cursor traverses execution history
Question: Why didn't Pac resize 0.5?
Your world is paused.
Resume
Stop
Why...
Undo
World.move Pac
Questions I've asked

Figure 3. The Whyline's answer shows a visualization of the runtime actions that prevented Pac from resizing.

Thus, to prevent errors, programming environments should help prevent these types of breakdowns. We have developed methods of studying programmers' work in order to determine how the programming language and environment might be changed to prevent breakdowns. We have recently focused on preventing breakdowns in debugging.

In order to see what tools might be useful, we performed two studies of both experts' and novices' programming activity [6] using the Alice programming environment (see Figure 2). We chose Alice for these studies because it simplifies the creation of programs by using drag-and-drop to place tiles of code into the code area and pop-up menus to choose parameters. This interaction prevents all type errors and syntax errors (see www.alice.org for more details).

We observed that *all* of the programmers' questions at the time of failure were one of two types: 32% were *why did* questions, which assume the occurrence of an *unexpected* runtime action, and 68% were *why didn't* questions, which assume the absence of an *expected* runtime action. Furthermore, 50% of all errors were due to programmers' false assumptions in the hypotheses they formed while debugging existing errors, resulting in the insertion of new errors that had to be debugged.

Design of the WhyLine

No existing programming environment allows users to ask these kinds of "why" questions. By analyzing the control flow graph of the programs, and annotating it with the complete history of all assignments and uses of properties' values, we are able to answer these questions directly in the Alice UI (see Figure 2) through the "Why did" and "Why didn't" menus. The submenus contain the objects in the world that were or could have been affected.

In addition to highlighting the relevant piece of code, we present a visualization of the answer to the question in Whyline—the Workspace that Helps You Link Instructions to Numbers and Events (see Figure 3).

We performed a user study comparing the Alice environment with and without the Whyline. Subjects were Master's students with programming experience ranging from beginning Visual Basic to extensive C++ and Java. Analyzing six situations that were identical across the two conditions, the Whyline significantly decreased debugging time from an average of 155 seconds per bug down to 20 seconds, which is a factor of 7.8. Furthermore, in the 90 minutes allotted, programmers with the Whyline completed 40% more tasks than those without. Full details are available in [7].

Conclusion

We are currently working on extending these ideas in many directions. We are investigating new domains in which to design more natural languages. We are extending the programming environment research to help with other parts of program analysis and creation.

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

References

1. Burnett, M., Yang, S., and Summet, J. A scalable method for deductive generalization in the spreadsheet paradigm. *ACM Trans. Computer-Human Interaction 9,* 4 (2002), 253–284.
2. Green, T.R.G. and Petre, M. Usability analysis of visual programming environments: A cognitive dimensions framework. *J. Visual Languages and Computing 7,* 2 (1996), 131–174.
3. Hoc, J.-M. and Nguyen-Xuan, A. Language semantics, mental models and analogy. J.-M. Hoc et al., Eds. *Psychology of Programming*. Academic Press. London, 1990, 139–156.
4. Johnson, J.A., Nardi, B.A., Zarmer, C.L., and Miller, J.R. Ace: Building interactive graphical applications. *Commun. ACM 36*, 4 (Apr. 1993). ACM, NY, 41–55.
5. Kehoe, C., Stasko, J., and Taylor, A. Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International J. Human-Computer Studies 54,* 2 (2001), 265–284.
6. Ko, A.J. and Myers, B.A. Development and evaluation of a model of programming errors. *IEEE Symposia on Human-Centric Computing Languages.* (Auckland, New Zealand, 2003), 7–14; www-cs.cmu.edu/~ajko/HCC2003.pdf.
7. Ko, A.J. and Myers, B.A. Designing the Whyline, a debugging interface for asking why and why not questions about runtime failures. In *Proceedings of 2004 Human Factors in Computing Systems (CHI04).* Vienna, Austria, Apr. 2004), 151–158.
8. Lieberman, H. The debugging scandal and what to do about it. *Commun. ACM 40,* 4 (Apr. 1997). Special section, 26–78.
9. Pane, J. A programming system for children that is designed for usability. Ph.D. thesis, 2002. Carnegie Mellon University, Pittsburgh, PA; www.cs.cmu.edu/~pane/thesis/.
10. Pane, J.F. and Myers, B.A. Usability issues in the design of novice programming systems. School of Computer Science Technical Report, CMU-CS-96-132 (Aug. 1996), Carnegie Mellon University, Pittsburgh, PA; www.cs.cmu.edu/~pane/tr96/.
11. Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM 24,* 9 (Sept. 1981), 563–573.
12. von Mayrhauser, A. and Vans, A.M. Program understanding behavior during debugging of large scale software. In *Proceedings of 7th Annual Workshop for Empirical Studies of Programmers.* (Alexandria, VA, 1997).

Brad A. Myers (bam@cs.cmu.edu) is a professor in the Human Computer Interaction Institute, School of Computer Science at Carnegie Mellon University, Pittsburgh, PA.
John F. Pane (jpane@rand.org) is an associate information scientist at RAND, Pittsburgh, PA.
Andy Ko (ajko@cmu.edu) is a Ph.D. student in the Human Computer Interaction Institute at Carnegie Mellon University, Pittsburgh, PA.

This research has been funded in part under NSF grants IRI-9900452 and IIS-0329090, and by the EUSES Consortium under NSF grant ITR-0325273. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

Notice of Intent to De-Charter

ACM is considering the de-charter of the following chapters due to inactivity. Members interested in revitalizing their chapter should contact Susan Wood, Local Activities Coordinator, wood_s@acm.org. ACM will terminate the chapters listed here after 90 days unless interested volunteers express a desire to reactivate their chapter and prepare acceptable revitalization plans.

Belgian ACM Chapter
Bulgarian ACM SIGCHI Chapter
Central Ohio ACM SIGCHI (BuckCHI) Chapter
Clear Lake Area ACM SIGAda Chapter
Czech ACM Chapter
Dallas/Ft. Worth ACM SIGCHI Chapter
Egypt ACM Chapter
Huachuca ACM SIGAda Chapter
Hungarian ACM Chapter
Huntsville ACM SIGAda Chapter
Hyderabad/India ACM Chapter
Italian ACM Chapter
Kaluga Region/Russian ACM SIGAPL Chapter
Los Angeles ACM SIGCHI Chapter
LVIV/Ukraine ACM Chapter
Mexico ACM SIGCAS Chapter
New York City ACM SIGCSE Chapter
Ottawa ACM SIGART Chapter
Rio Grande ACM Chapter
Rochester ACM Chapter
Rome/Italy ACM SIGAPL Chapter
Southeast Michigan ACM SIGART Chapter
Tampa Bay ACM Chapter
Toronto ACM SIGAPL Chapter
Washington D.C. ACM SIGAda Chapter
Westchester Fairfield ACM Chapter
Yemen ACM Chapter
Yogyakarta-Indonesia ACM Chapter
Yokohama/Japan ACM SIGAda Chapter
Oktibehha ACM SIGGRAPH Chapter