Natural Programming:
Project Overview and Proposal

Brad A. Myers

January, 1998
CMU-CS-98-101
CMU-HCII- 98-100

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

bam@cs.cmu.edu
http://www.cs.cmu.edu/~NatProg

Abstract

End-users must write programs to control many different kinds of applications. Examples include multimedia authoring, controlling robots, defining manufacturing processes, setting up simulations, programming agents, scripting, etc. The languages used today for these tasks are usually difficult to learn and are based on professional programming languages. This is in spite of years of research highlighting the problems with these languages for novice programmers. The Natural Programming Project is developing general principles, methods, and programming language designs that will significantly reduce the amount of learning and effort needed to write programs for people who are not professional programmers. These principles are based on a thorough analysis of previous empirical studies of programmers, as well as new studies designed to discover the most natural programming paradigms. Our proposed research is to extend these results, and apply them to different domains. The result will be new programming languages and environments that are demonstrably superior for users.

Keywords: End-User Programming, Programming Languages, Code, Multimedia Authoring, Robot Control, Process Control.

One Page Executive Summary

The Natural Programming Project is studying ways to make learning to program significantly easier, so that more people will be able to create useful, interesting and sophisticated programs. The goals of this project are to study how non-programmers reason about programming concepts, and then to create one or more new programming languages and environments that take advantage of these findings. It is somewhat surprising that in spite of 30 years of research in the areas of Empirical Studies of Programmers (ESP) and Human-Computer Interaction (HCI), the designs of new programming languages have generally not taken advantage of what has been discovered. For example, the new Java and JavaScript languages use the same mechanisms for looping, conditionals and assignments that have been shown to cause many errors for both beginning and expert programmers in the C language. A thorough investigation of the ESP and HCI literature has revealed many results which can be used to guide the design of a new programming system, many of which have not be utilized in previous designs. However, there are many significant "holes" in the 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. Many new systems are object-oriented (C++, Java), but systems aimed at novice programmers tend to be event-based instead (Visual Basic, HyperTalk), and the language research community is studying functional styles. Or maybe the old imperative style is easiest for beginning programmers? We will perform user studies to investigate this question. Another issue is that most of the prior research has studied people using existing languages, and so there is little information about how people might express various concepts if not restricted by an existing language. Our pilot studies suggest that some interesting results might arise from investigating this question. For example, participants usually expressed iterations implicitly, by operating on sets of objects, as in: "When PacMan eats all of the yellow balls he goes to the next level." Another observation is that the usual case was often expressed first with exceptions afterwards, as in: "When you encounter a ghost the ghost should kill you. But if you get a little pill you can eat them." All of these capabilities are not provided by any of today's languages.

We propose a set of studies to see how people naturally express programming concepts, and then further studies comparing different proposed designs for language features. The result will be new knowledge which can be used as guidelines for the design of new languages. We will then incorporate these findings into one or more new programming languages and environments. We plan to target the many domains in which it is useful or necessary for people who are not professional programmers to create programs to control the computer. The first domain will be a new programming language for children, that will make it easier for the children and their teachers to create their own games and educational software. Other possible domains include: "tailorable" systems (where the end-user customizes the system by adding new functionality), general scripting and macros in a direct manipulation system (often to automate repetitive tasks), creating CGI scripts for world-wide-web pages, simulation setup, multimedia authoring, controlling robots, process control for manufacturing, and programming agents. We will exploit modern technologies to simplify the programming process, including direct manipulation and demonstrational techniques, programming environment technologies like structure editors, and mechanisms for encapsulation to create reusable components.

Introduction

The goals of the Natural Programming Project in the Human-Computer Interaction Institute at Carnegie Mellon University are to develop new methods, techniques and tools that will make it significantly easier for people to write programs. We are particularly targeting applications where people who are not professional programmers are expected to write the programs. By studying how non-programmers *naturally* express the tasks that they want the computer to do, we believe we can create more natural programming languages, which will then be easier to learn. The design will be based on principles from Human-Computer Interaction and the Empirical Studies of Programmers. The results will be applicable to a wide variety of audiences, including programming languages for children, teachers, office workers, military planners, world-wide-web authors, etc. In fact, all computer users would benefit from the ability to automate their repetitive tasks and customize their applications through scripting.

This document uses the term “authoring” to emphasize that the code is being written by people who are not professional programmers. “Authoring” ranges from writing small scripts in Visual Basic all the way up to creating new applications. We are exploring techniques which will make the entire range of authoring easier to learn and more effective.

Motivation

Programming

Direct manipulation user interfaces have enabled millions of people to use computers. Yet it is well known that direct manipulation has limitations: it is not effective for repetitive tasks, for handling large numbers of objects, or for dealing with abstractions. In these cases, some kind of programming is needed, often called *scripting* or *macros*. Scripting is available in some direct manipulation applications. Examples of these *End-User Programming Languages* include Visual Basic for many Microsoft applications, Lingo for MacroMedia Director, AutoLisp for AutoCAD, HyperTalk for HyperCard, “macro” languages for spreadsheets, etc.

We believe that the need for programming by the general computer user will only increase. Workers today deal mainly with digital information. An estimated 80% of all salaried workers will work with computers by the year 2000, including almost all office workers [Billingsley 1995]. People whose jobs consist mainly of manipulating information on computers are well-motivated to create custom programs to help them with the processing, because it will make them more productive.

Teachers are another group of people who will need to use computers more extensively. They must prepare lessons that are individualized to each classroom and even to each student, and they do this today by picking and choosing from their text books, and by creating much of their own material and handouts. If teachers are to use computers effectively in education, then the teachers themselves will similarly need to be able to refine existing *computerized* material and create their own material, so that it will be geared to their particular needs and their particular classes.

Another motivation for providing programming capabilities to end users is that the popular direct manipulation techniques are beginning to break down. Direct manipulation works fine for hundreds of objects, but now that multi-gigabyte disks hold tens of thousands of files, and the World-Wide-Web contains 100 million pages, the objects of interest can no longer be simply viewed and pointed at. People are frustrated that the direct manipulation programs are typically not extensible or programmable (what you see is *all* you get), and the interfaces that are extensible use programming languages and querying techniques are too complicated for the average user. Computers are powerful and can do virtually anything that a user might want, but commercial applications, even with their hundred of commands, do not do *specifically* what is needed by each individual user. Only through programming can people customize applications to meet their specific individual needs.

Our goal is to study the human side of programming in order to create new languages that are flexible while still being easier to learn and use. Why is this necessary? More than a decade ago, Allen Newell pointed out:

> "Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. ... The human and computer parts of programming languages have developed in radical asymmetry." [Newell 1985]

This situation still holds. There are substantial gaps in the knowledge about how to make programming languages effective for people, and how to apply this knowledge to the design of new programming languages. The Natural Programming Project will help to fill these in.

Natural

We define *natural* as “faithfully representing nature or life,” which here implies that it works in accordance with the way people expect. By “natural programming” we are aiming for the language to work in the way that people who do not have programming experience would expect. Why would this make the programming easier? One way to define programming is the process of transforming a mental plan in familiar terms into one that is compatible with the computer [Hoc 1990]. The closer the language is to the user’s original plan, the easier this refinement process will be. This 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 describe directness as the distance between one’s goals and the actions required by the system to achieve those goals [Hutchins 1986]. Reducing this distance makes systems more direct, and therefore easier to learn. User interface designers and researchers have been promoting directness at least since Shneiderman identified the concept in 1982 [Shneiderman 1983], but it has not even a consideration in most programming language designs. Green and Petre also argue in favor of directness, which they call *closeness of mapping:* “The closer the programming world is to the problem world, the easier the problem-solving ought to be.... Conventional textual languages are a long way from that goal.” [Green 1996, p. 146]

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 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 [Nielsen 1993, p. 126]. One of Hix and Hartson’s usability guidelines is *Use Cognitive Directness*, which means to “minimize the mental transformations that a user must make. Even small cognitive transformations by a user take effort away from the intended task” [Hix 1993, p. 38]. Conventional programming languages require the programmer to make tremendous transformations from the intended tasks to the code design. For example, to add a set of numbers uses 3 kinds of parentheses and 3 kinds of assignment operators in 5 lines of C code, whereas a single “SUM” operator is sufficient in a spreadsheet [Green 1996].

Expected Benefits

Having a more natural programming language and environment is expected to **increase the number of people who will be able to program**. This will allow domain specialists to more easily create their own programs, without needing an outside programmer to do it for them. Less training should be required to learn to use these systems, which will make it cheaper and faster to become productive.

A more natural programming language is also expected to **make the programming process faster and more efficient**, since the language will be better suited to the tasks. The code should also be more readable and understandable, since we will apply the lessons learned from studies of program comprehension [Fitter 1979; Baecker 1986]. Since any useful program is going to be too large for the author to remember all of, going back and re-reading (and understanding) existing parts is an important part of program generation and modification.

Finally, we expect that the generated **programs will be more correct** than with today's languages. Human-computer interaction principles will be applied to *minimize errors* and provide good *feedback*. The code in the new language will be easier to generate and read, and we will incorporate new methods for testing and debugging the software that have proven to be effective but have not been generally available [Fry 1997; Myers 1997a].

Preliminary Results

Background Research

Our first step in thinking about the design of new easily-learned languages was to thoroughly study the Empirical Studies of Programmers (ESP) and Human-Computer Interaction (HCI) literature. It is somewhat surprising that in spite of 30 years of research in these areas, the designs of new programming languages have generally not taken advantage of what has been discovered. We cataloged many results which can be used to guide the design of a new programming system [Pane 1996]. For example:

- One way to ease the entry into programming is to capitalize on the beginner's knowledge about the **world. Many languages are based on a metaphor, which should be drawn from a concrete real-**world system that is familiar to the user audience [Smith 1994].

- When they are stumped, beginners will attempt to transfer knowledge from other domains even if they are not appropriate [Hoc 1990]. This is a problem when the language uses words and symbols in ways that are different from English or math. For example, “AND” is often read to mean “THEN” as in: “We went to the store **and** bought milk,” whereas in computers, AND is always used between two things that must both be true at the same time. People often use “AND” when a computer would require the use of “OR,” as in: “All people whose names begin with ‘A’ **and** ‘B’ should be in the first line.” Another problematic example is that “a = a+3” makes no sense if read as in mathematics. These kinds of features should be avoided in the new language.
- A very low-level language with many simple primitives requires the user to synthesize higher-level operations. This is one of the great difficulties in programming [Lewis 1987]. When there are many different choices, more planning is required, and this increases the likelihood of backtracking and revision, which slows the programmer [Gray 1987]. Therefore, the language should provide high-level operations.
- The object-oriented style seems to be harder to learn for novice programmers, and a full inheritance hierarchy has been shown to be too complex for novices, but a fixed two-level inheritance hierarchy is understandable [Pausch 1992].

However, there are many significant gaps in the knowledge about how people reason about programs and programming, and how languages can be made more effective. In particular:

- What programming paradigm works best for non-programmers? Professional languages like Java and C++ are object-oriented, but most novice languages, like Visual Basic and HyperTalk are not. And, should the language be textual or graphical?
- How can difficult constructs like iterations and conditionals be minimized and made easier?
- What is the tradeoff between ease of use and correctness? In particular, what is the role of type checking?
- How should abstractions, such as including variables, procedures and modules, be presented? To what extent do non-programmers focus only on the concrete examples?

- How can the reuse of procedures, modules and other components be facilitated?
- What terminology and syntax should be used? HyperTalk, AppleScript, and other recent languages from Apple have used a verbose style with extra optional words like “the” and “set.” Is this better than the more conventional language design using a terse syntax and special symbols like `=  :=  ==  {}[]()"'` ? Are AND, OR, and NOT the best words for expressing Boolean concepts?
- What are effective metaphors for various domains?
- What is the role of the environment in overcoming problems in the language? For example, syntax editors can overcome some problems with the language syntax, and intelligent tutors might help with learning the language.

New Studies

To start filling in these gaps, we are conducting new empirical studies on some important missing areas. These studies use people who do not already know how to program, but who have familiarity with a variety of computer applications. The studies are designed to discover how the people think and talk about programming concepts.

Pilot studies were conducted by asking a number of students to describe how they would make PacMan move about the screen, eating dots and killing or being killed by monsters. Among the observations from the pilot study are:

- Much of the control was expressed in an “event language” (also called the “production language”) style, with rules to control behaviors. This result is already reflected in some of today’s end-user programming languages. The event-based style used by Visual Basic, Lingo for Director, and HyperTalk for HyperCard, is a form of rule-based style, since the code is of the form “if this event happens, then execute this code.”
- The students preferred to express the general case first, and then later modify it with exceptions. For example, “When you encounter a ghost, the ghost should kill you. But if you get a little pill you can eat them.” This is in contrast to conventional languages that generally require the conditional to be set up in advance using “ANDs,” “NOTs” and “ORs,” forcing the user to think about all the cases first, and resulting in a complicated Boolean expression.
- The students expected objects to be moving as their normal behavior, and wrote commands that would alter the motion. For example, “If PacMan hits a wall, he

stops." This is in contrast to conventional languages and environments where to make something move requires setting its position at each clock timer tick.

- Iterations were usually expressed implicitly, by operating on sets of objects. For example, "When PacMan eats all of the yellow balls he goes to the next level." This is instead of using any form of iteration or explicit counting, as would be required in most programming languages.

Many researchers have identified control structures as a common area of difficulty for novice programmers [Hoc 1990]. The observations noted above from the pilot study partially *eliminate* control structures by making loops and conditionals implicit. Thus, we conjecture that creating a new language that supports these natural tendencies will be easier to learn.

New Textual Language

We expect the new languages to be primarily textual, augmented by a supporting programming environment. The paradigm, format, and syntax of the language will be based on the results of the studies, and have not yet been resolved.

Many people argue that programming is difficult because it requires the precise use of a textual language, and that a system that eliminates language will be inherently easier to use [Smith 1994]. However, to date nobody has been able to prove that a visual language is superior to textual languages for all tasks [Green 1991]. To the contrary, often textual languages are superior to visual languages [Green 1992]. It seems that visual languages might be better for small tasks, but often break down for large tasks, and we want to make sure that the proposed language will be appropriate for creating complete applications. The good news is that it is not always difficult to learn to program in a textual language. In fact, the most successful end-user programming system is the spreadsheet, which is text-based [Nardi 1993].

New Metaphors

One of the prominent themes in the prior research is the importance of a familiar concrete model for the computational system [Lewis 1987]. One way to obtain these critical attributes of concreteness and familiarity is to use a well-known real-world system as a metaphor for the computational machine. Many previous research systems used metaphors, such as the "turtle" in Logo.

Our proposed system will also use a metaphor. Since our experiments suggest that a production-system style of programming would be more natural, we searched for a metaphor that would make this more concrete, as well as addressing the other difficult aspects, such as control structures [Hoc 1990]. After trying many possibilities, the idea we are currently exploring is to use a *card game*. This metaphor has many compelling attributes. First, it is a familiar concept for the target audience. Second, the basic elements of a card game (cards, players, hands, piles, table, face values, suits, etc.) are concrete and can be shown graphically, so they do not require abstract thinking or imagination. Finally, there seems to be a useful mapping between a card game and the essential concepts of programming, so that programs can be represented in a complete and consistent manner within the metaphor. Thus we hope many of the difficult aspects of programming will be easier because they will be presented more concretely, allowing beginners to quickly get started in programming by transferring their knowledge from card games.

As an example of how concepts familier to programmers are handled by the card game metaphor, the production rules might be written onto rule cards, and then the order of the cards in a hand would serve as a concrete representation for the **order of evaluation**. **Disabling** of some rules, which is important to support the different **modes** of a system, might be represented concretely by placing the rule cards face down on the table. **Data** might be represented concretely on data cards, and then **allocation** and **deallocation** might be just picking up and discarding cards from a pile. **Types** might be represented by different suits, and **processes** or **threads** by different players. **Information hiding** might be enforced by not allowing a player to see cards in another player's hand, and **message passing** might be performed concretely by moving a card from one player's hand to another's. **Distributed computation** might be supported by allowing the players to represent processes that might be executing on different machines. **Information sharing** might be provided by allowing any player to see data cards which are face up on the table. The table then acts like the "blackboard" from AI architectures and the "tuple space" of Linda [Gelernter 1992].

Development Environments

To minimize some of the difficulties with textual languages, we will create an editor and a run-time environment (interpreter and debugger). For example, many of the problems of textual languages can be attributed to problems with punctuation and other syntactic embellishments that are used to allow the compiler to efficiently and correctly parse a stream of text. Our new language will eliminate the need for most of this punctuation by using structure-editor or form-based technology, which provides slots for the unambiguous placement of program elements. Programming systems that are easy to learn and easy to use have a direct manipulation front end which significantly reduces the amount of necessary scripting. For example, in Visual Basic, users can place widgets (also called “controls”) using the mouse and set their properties using dialog boxes. One focus of our previous work has been how to *extend the range* of what can be performed by direct manipulation, by allowing more *behaviors* to be specified by demonstration [Myers 1992a]. This means that when the author wants something dynamic to happen at run-time, it can be demonstrated at design time. One of the frustrating things about tools like Visual Basic is that it is very easy to change the properties of an object at design time, by just selecting the object and moving it or bringing up its property sheet. However, this gives no clue about how to write code to perform these actions at run time in a program. Our system will be “self-disclosing” [DiGiano 1995] so that every time the user performs some action by direct manipulation or by demonstration, the system will show some example code that will perform the same action. This may also help the users learn the programming language, and will provide templates and code snippets which they can edit.

Domains

There are many domains where the end users would benefit from programming capabilities, and having a better language for programming would be helpful in all of these. We believe that the general principles and concepts we are developing can be applied in each of these domains to create more effective domain-specific programming environments.

- **Programming for children:** Our initial environment will be a new programming language for children, which makes it easier for children and their teachers to create games and educational simulations.
- **“Tailorable” systems:** We define *tailoring* as the activity of modifying a computer application within the context of its use. Tailoring is therefore distinguished from both use and development, but it borrows terminology from each: tailoring is further development of an application during its use to adapt it to needs that were not accounted for in the original design. Tailorability is widely assumed to be a key requirement for the design of groupware systems, due to rapidly changing work

situations. Other kinds of applications benefit from tailorability as well. A well-defined and usable language is key to effective tailorability.

- **General scripting:** There are many situations in which repetitive actions must be performed, and writing a script or macro to perform them automatically would make the user more effective. Such scripting languages are available in the Unix shell and in modern spreadsheets and word processors (most Microsoft products use Visual Basic as the scripting language). One example use for scripting is to transform data from one format to another. Often, users will find information on the world-wide-web or in a database, and need to copy and reformat it to enter it into some other database or file. Being able to write small reusable scripts for this would be helpful.
- **World-Wide-Web:** Interactive tools like Microsoft Frontpage or Adobe PageMill make creating static pages with text and graphics quite easy. However, for pages that allow users to fill in fields, or where the next page to display is based on data supplied by the user, the author must usually write a script using a language such as PERL, Java, Javascript or Visual Basic. Having a more usable and learnable language for processing this input would empower a wider range of people to construct dynamically generated pages.
- **Simulation setup:** Interactive tools like ModSAF [Calder 1993] allow some parts of a simulation to be specified using maps and dialog boxes, but other parts still need to be specified by writing programs. This applies to military simulations as well as simulations of physical processes.
- **Multimedia authoring:** Tools like Macromedia's Director require that much of the behavior of objects be specified by writing code in the scripting language. Multimedia that allows the viewer to interact with the objects in interesting ways will always require the author to write scripts.
- **Educational software:** For educational software to be effective, the teachers themselves must be able to refine existing material and create their own educational material, so that it will be geared to the needs of their particular classes. Since educational software is usually interactive, with small games, quizzes, and user-selectable options, this means that many teachers will need to program.
- **Controlling robots and process control:** Many manufacturing and robot systems require the user to program the process to be followed. The processes often include

logic to test various conditions and branch to different actions depending on values from sensors.

- **Intelligent Agents:** Agents are emerging as an important programming paradigm. Many agents are designed to be controlled or configured by the user, and often a scripting or programming language is necessary.

Deliverables

The first result of the Natural Programming Project will be new, fundamental knowledge about effective ways for non-programmers to author programs. This will include an interpretation and organization of prior research about novice programmers, which will guide the design of our systems, and help other designers to apply this knowledge in their work. Through a set of new studies that investigate aspects of novice programming that have not yet been fully covered, we will discover natural tendencies that beginners exhibit when describing solutions to programming tasks. These results will be disseminated in papers and technical reports, copies of which will be sent to all sponsors.

In addition to papers, we will develop prototype implementations. Our group has had a long history of developing research prototypes that are released for widespread use. For example, our Amulet toolkit [Myers 1997b] has been downloaded over 10,000 times, and has been used for hundreds of research and commercial products. Our prior Garnet toolkit [Myers 1990b] is also widely used. Similarly, we expect to make the results of the proposed research available for general use. Our sponsors will be able to internally evaluate any prototypes developed during this research, and will have the opportunity to download the source code for the software.

An important open question is whether there will be one language and environment that will be appropriate for multiple domains, or whether the principles and methods will be used to create a *family* of domain-specific languages. The first language and environment to be developed will be a new programming language for children, and some of the human factors studies will address whether the lessons learned from that language also apply to adults. Depending on funding, other domains will be investigated to see what can be transferred from this first project to create an appropriate language and environment for each other domain. We expect that there will be a great deal of transfer, but further research is required.

Related Work

It is somewhat surprising that the designs of new programming languages have generally not taken advantage of most of the human-factors results. For example, the newly popular Java and JavaScript languages use the same mechanisms for looping, conditionals and assignments that have been shown to cause many errors for both beginning and expert programmers in the C language. Other modern languages, such as ML, have been designed to explore technical issues such as provability and compilability, and have intentionally ignored issues of learnability and usability. Even programming systems designed for end users have ignored issues of usability for the language itself. For example, Visual Basic has an excellent environment and editor, but the programming language itself is still based on the original Basic and has many well-known problems. Scripting languages in other tools, such as HyperTalk for HyperCard and Lingo for Director have many problems, including inconsistencies, reliance on difficult control constructs, insufficient debugging support, lack of scalability, etc.

The most successful and widely used language for children has been Logo [Papert 1971]. The language borrowed heavily from Lisp, but the syntax was redesigned to be easier to learn and read. Unfortunately, this process resulted in a language that has unusual punctuation, cryptic names and symbols for commands and operators, and most Logo implementations have an environment that is difficult to use Logo is also not powerful enough to create highly interactive graphically-rich programs like the commercial software that children use.

The most successful end-user programming system to date is the spreadsheet, due in part to its familiar and effective metaphor of financial tables [Nardi 1993]. Unfortunately, the spreadsheet metaphor lacks generality, and thus many kinds of programs are not well-suited to implementation in a spreadsheet. Many programming environments for end-users have used visual or graphical programming languages (see [Myers 1990a] for a survey), but these have usually proven quite limiting [Green 1992].

Agentsheets [Repenning 1993] and its descendants such as Cocoa (formerly called KidSim) [Smith 1994], are a family of programming environments for end-users, based on autonomous communicating agents in a two-dimensional grid-based world. These systems combine graphical rewrite rules and programming by example to simplify programming. However, one quickly encounters the limitations of the rewrite rules and the grid: it is difficult or impossible to generalize the graphical rewrite rules for complex situations; and the grid prevents the implementation of many interesting physical phenomena that require continuous movement such as gradients of molecules, optical paths, collisions of rigid objects, velocity and acceleration of projectiles, etc.

Gentle Slope Systems

The proposed research is closely aligned with the concept of “Gentle Slope Systems” [Dertouzos 1992] [Myers 1992b] which are systems where for each incremental increase in the level of customizability, the user only needs to learn an incremental amount. This is contrasted with most systems which have “walls” where the user must stop and learn many new concepts and techniques to make further progress (see Figure 1). We propose to use direct manipulation and demonstrational techniques to lower the initial starting point (so users can get useful work done immediately), and create a language that is self-disclosing and easy to learn so the number and height of the walls is minimized, if they cannot be eliminated entirely.

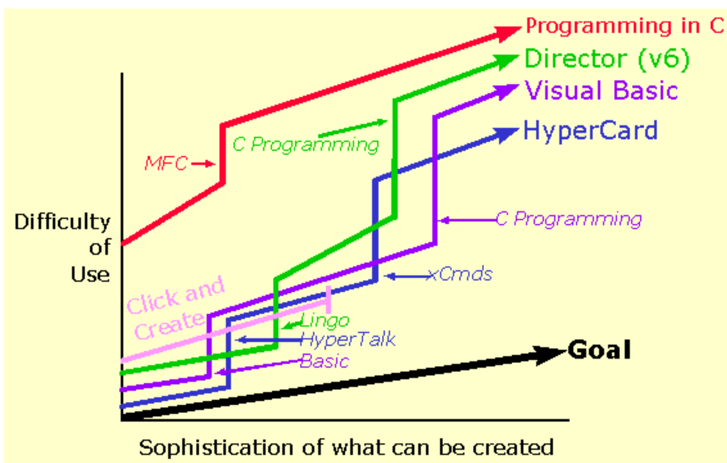

Figure 1: The intent of this graph is to try to give a feel for how hard it is to use the tools to create things of different levels of sophistication. For example, with C, it is quite hard to get started, so the Y intercept is high up. The vertical walls are where the designer needs to stop and learn something entirely new. For C, the wall is where the user needs to learn the Microsoft Foundation Classes (MFC) to do graphics. With Visual Basic, it is easier to get started, so the Y intercept is lower, but Visual Basic has two walls—one when you have to learn the Basic programming language, and another when you have to learn C. Click and Create is a menu based tool from Corel, and its line stops because it does not have an extension language, and you can only do what is available from the menus and dialog boxes.

Conclusion

In conclusion, we expect that the research on Natural Programming will result in new knowledge about how to make programming easier for non-programmers, along with new languages and environments for various domains that will embody these results and make it easier to author interesting, useful and effective programs. Because they will be easier to learn, more people will be able to author programs, which will allow domain specialists with little programming experience to automate their tasks and therefore be more effective. Programming should be faster, more efficient, and more correct because the language and environment will be matched to the users' tasks. We hope to raise support so we can investigate the application of Natural Programming to many different domains where end-users would benefit from much better programming capabilities.

Acknowledgements

The students working on the Natural Programming Project include John Pane (Ph.D. student) and Chotirat ("Ann") Ratanamahatana (BS student). Some of this project description was adapted from John Pane's thesis proposal. Funding for this project is still pending.

References

[Baecker 1986] Ronald Baecker. "Design Principles for the Enhanced Presentation of Computer Program Source Text." Proceedings of CHI'86 Conference on Human Factors in Computing Systems. M. Mantei and P. Orbeton. 1986: Boston, ACM. pp. 51-58.

[Billingsley 1995] Pat Billingsley. "Hard Test for Soft Products," *SIGCHI Bulletin.* 1995. **27**(1). p. 10.

[Calder 1993] R. B. Calder, J. E. Smith, A. J. Courtemanche, J. M. F. Mar and A. Z. Ceranowicz. "ModSAF Behavior Simulation and Control," *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representational,* Orlando, FLA, 1993. pp. 347-356.

[Dertouzos 1992] Mike Dertouzos and et al. *ISAT Summer Study: Gentle Slope Systems; Making Computers Easier to Use.*

[DiGiano 1995] Chris DiGiano and Michael Eisenberg. "Self-Disclosing Design Tools: A Gentle Introduction to End-User Programming." *Proceedings of DIS'95 Symposium on Designing Interaction Systems.* 1995: pp. 189-197.

[Fitter 1979] M.J. Fitter and T.R.G. Green. “When Do Diagrams Make Good Computer Languages?,” *International Journal of Man-Machine Studies.* 1979. **11** pp. 235-261.

[Fry 1997] Christopher Fry. “Programming on an Already Full Brain,” *Communications of the ACM.* 1997. **40**(4). pp. 55-64.

[Gelernter 1992] David Gelernter and Nicholas Carriero. “Coordination Languages and their Significance,” *Communications of the ACM.* 1992. **35**(2). pp. 96-107.

[Gray 1987] W. Gray and J.R. Anderson. “Change-Episodes in Coding: When and How Do Programmers Change Their Code.” *Empirical Studies of Programmers: Second Workshop.* G. M. Olson, S. Sheppard and E. Soloway. 1987: Norwood, NJ, Ablex. pp. 185-197.

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

[Green 1996] T.R.G. Green and M. Petre. “Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework,” *Journal of Visual Languages and Computing.* 1996. **7**(2). pp. 131-174.

[Green 1991] T.R.G. Green, M. Petre and R.K.E. Bellamy. “Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the ‘Match-Mismatch’ Conjecture.” *Empirical Studies of Programming: Fourth Workshop.* J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson. 1991: New Brunswick, NJ, Ablex Publishing Corporation. pp. 121-146.

[Hix 1993] Deborah Hix and H. Rex Hartson. *Developing User Interfaces; Ensuring Usability Through Product & Process*. New York, John Wiley & Sons, Inc. 1993.

[Hoc 1990] Jean-Michel Hoc and Anh Nguyen-Xuan. “Language Semantics, Mental Models and Analogy.” *Psychology of Programming.* J.-M. Hoc, T. R. G. Green, R. Samurçay and D. J. Gilmore. 1990: London, Academic Press. pp. 139-156.

[Hutchins 1986] Edwin L. Hutchins, James D. Hollan and Donald A. Norman. “Direct Manipulation Interfaces,” *User Centered System Design*, Hillsdale, New Jersey, Lawrence Erlbaum Associates. 1986. pp. 87-124.

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

[Myers 1990a] Brad A. Myers. “Taxonomies of Visual Programming and Program Visualization,” *Journal of Visual Languages and Computing.* 1990a. **1**(1). pp. 97-123.

[Myers 1992a] Brad A. Myers. "Demonstrational Interfaces: A Step Beyond Direct Manipulation," *IEEE Computer.* 1992a. **25**(8). pp. 61-73.

[Myers 1997a] Brad A. Myers, Alan Ferrency, Rich McDaniel and Roger Dannenberg. *Debugging Interactive Applications*.

[Myers 1990b] Brad A. Myers, *et. al.* "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces," *IEEE Computer.* 1990b. **23**(11). pp. 71-85.

[Myers 1997b] Brad A. Myers, *et. al.* "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering.* 1997b. **23**(6). pp. 347-365.

[Myers 1992b] Brad A. Myers, David Canfield Smith and Bruce Horn. "Report of the `End-User Programming' Working Group," *Languages for Developing User Interfaces,* Boston, MA, Jones and Bartlett. 1992b. pp. 343-366.

[Nardi 1993] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA, The MIT Press. 1993.

[Newell 1985] Allen Newell and Stuart K. Card. "The Prospects for Psychological Science in Human-Computer Interaction," *Human-Computer Interaction.* 1985. **1**(3). pp. 209-242.

[Nielsen 1993] Jakob Nielsen. *Usability Engineering*. Boston, Academic Press. 1993.

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

[Papert 1971] Seymour Papert. *Teaching Children Thinking*. Cambridge, MA, MIT. AI Memo No. 247 and Logo Memo No. 2, 1971.

[Pausch 1992] Randy Pausch, Matthew Conway and Robert DeLine. "Lesson Learned from SUIT, the Simple User Interface Toolkit," *ACM Transactions on Information Systems. ACM Transactions on Information Systems.* 1992. **10**(4). pp. 320-344.

[Repenning 1993] A. Repenning. "Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments." *Proceedings of INTERCHI '93, Conference on Human Factors in Computing Systems.* 1993: Amsterdam, pp. 142-143.

[Shneiderman 1983] Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer. IEEE Computer.* 1983. **16**(8). pp. 57-69.

[Smith 1994] David Canfield Smith, Allen Cypher and Jim Spohrer. "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM.* 1994. **37**(7). pp. 54-67.