Invisible Programming

Keynote Speech

Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The topic of this conference is Visual languages, but I want to discuss computer programming using a technique where there is no apparent language at all; the language is (mostly) invisible. Here, the user sees the results of the program execution and the data the program is operating on, but the program itself is not shown. The program is specified by demonstrating the operations that should be performed using example data. Therefore, these systems are called "programming-by-example" or "demonstrational" interfaces. This paper presents an overview of this intriguing idea, and presents a survey of existing systems and an agenda for future research.

Introduction

This paper discusses a new style of user interface where the user gives an *example* of the desired operation, and the system generalizes to construct a general-purpose procedure. These are called *Demonstrational Interfaces*, because the user is demonstrating to the system what should be done. When a demonstrational interface provides true programming capabilities, then it is called *Programming by Example*.

This paper more formally defines demonstrational interfaces and related terms, and discusses why I think they are important. Next, a survey of existing uses of this technology is presented. Finally, some areas for future work are discussed.

Definitions

Demonstrational user interfaces provide concrete examples on which the user operates, rather than requiring the user to deal with abstractions such as variables and control structures.

There are two ways that demonstration can be used in user interfaces. One is that the user provides examples and the system guesses (or "infers") how the examples should be generalized to create something that is more general-purpose. At one extreme are systems that try to generate computer programs from examples of input and output [14]. More successful programs limit the inferences to a specific domain. For example, Peridot [7] is a graphical editor that creates user interface components by generalizing from the specific examples for parameters. In Peridot, the user can define a menu using a particular set of strings, but the system creates a procedure that works for *any* list of strings.

The second kind of demonstrational interface does not use inferencing, but allows the user to have example values available while operations are executed. These operations affect the example values in addition to being recorded. The EMACS editor [17] uses this technique to allow the user to create macros simply by going into a special mode and executing the editor commands normally. The macros can then be used later with different text. This use of examples can be differentiated from conventional testing and debugging because the examples are used *during* the development of the code, not just after the code is completed.

Some demonstrational interfaces do not provide full *programming*. To be considered programmable, the system must include the ability to handle variables, conditionals and iteration, at least implicitly. The EMACS macros mentioned above do not provide programming, whereas Peridot does.

Demonstrational interfaces that provide programming capabilities are called *Example-based Programming* [10]. When inferencing is used, these are called *Programming-by-example*. This is often called "automatic programming" and has generally been an area of Artificial Intelligence research. *Programming-with-Example* systems, however, require the programmer to specify everything about the program (there is no inferencing involved), but the programmer can work out the program on a specific example. The system executes the programmer's commands normally, but remembers them for later re-use.

Finally, the term *Intelligent Interfaces* refers to any user interface that has some "intelligent" or AI component. This includes demonstrational interfaces with inferencing, but also other interfaces such as those using natural language.

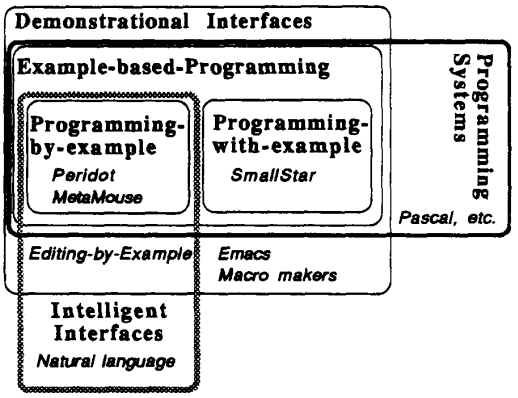

Figure 1: A taxonomy of interfaces. The systems named in italics are discussed in the text.

Figure 1 shows how these categories create a taxonomy for classifying systems.

Motivation for Demonstrational Interfaces

It is well known that people are much better at dealing with specific examples than with abstract ideas. A large amount of teaching is achieved by presenting important examples and having the students do specific problems. This helps them understand the general principles. It is well know that people make fewer errors when working out a problem on an example as compared to performing the same operation in the abstract, as in conventional programming [16]. The programmer does not need to try to keep in mind the large and complex state of the system at each point of the computation if it is displayed for him on the screen [15].

In particular, using demonstrational techniques in user interfaces has two significant advantages:

- It can provide programming capabilities to users without requiring any special programming knowledge, and,
- It can make the user interface more efficient and easier to use.

These are discussed further in the next sections.

Programming Capabilities

The vast majority of people who use computers do not know how to write conventional computer programs. However, special application-specific languages, such as Lotus 1-2-3 for spreadsheets, are used by large numbers of people to customize their applications. For other applications, there is no natural way to provide programming capabilities. This is especially true for graphical applications where there is usually no textual representation of the interactive commands executed with a mouse or other pointing device.

Demonstrational techniques can be used to provide these programming capabilities without requiring the user to learn a programming language. The user performs the actions in the usual way, and they are recorded for later re-use. Variables, loops, conditionals and other programming features can then be added to the generated scripts either automatically by the system using inferencing, or explicitly by the user. This technique has been successfully demonstrated for desktops [4] and for user interface construction [7]. Note that although the user creates a program, the code may be hidden, so the program itself is "invisible."

Easier to Use

When the system can infer what the user's intentions are, it can save the user from having to perform a number of steps. Most direct manipulation systems provide the user with a small number of simple and direct operations, out of which the desired high-level effects can be constructed. For example, drawing packages allow the user to change the position and size of individual objects or groups of objects. However, there are rarely tools that help with higher-level effects like getting objects to be evenly spaced. A demonstrational system might watch the user as the first few objects were moved, and automatically infer this high level property. It could then move the rest of the objects appropriately so the user would not have to. Using inferencing in this way has been successful in limited domains, such as user interface toolkit construction [7] and creating simple drawings and animations [6].

In addition to helping the user avoid repetitive actions, demonstrational techniques can also be used to infer semantic properties of the objects. For example, in a user interface, the height of a rectangle might be used as an indicator for some value. Rather than requiring the user to type the formulas that connect the rectangle size with the controlling variable, the user might simply draw the rectangle in its two sizes, and the system would then automatically create the code, as in Peridot [7].

For most of these functions, it would be possible to provide the user with a command that performed the same action that the system infers from the demonstration. The advantages of providing inferencing instead of extra commands are that:

- This might significantly decrease the number of commands and therefore make the system easier to use and learn.
- To combine the commands appropriately may require knowledge of programming techniques that the users do not have.
- The user may not know the correct high level semantic property that will give the desired result, whereas the

system may be able to tell which is appropriate from the examples.

- The demonstrational system can be set up to always try to determine a high level relationship, but with commands, the user might forget to apply the appropriate command.

Survey

The next sections discuss some systems that have demonstrational interfaces.

Systems Without Inferencing

Perhaps the simplest demonstrational interfaces are keyboard macros for text editors such as EMACS [17]. Here, the user goes into program mode, executes a number of commands, and then leaves program mode. The commands execute normally, and are also saved so they can be replayed. This idea has been used in simple transcript programs for the Macintosh user interface, such as MacroMaker from Apple. Unfortunately, it is less successful here because the transcripting programs are not tied to a particular application and therefore can only save raw mouse and keyboard events. Often, macros will not work correctly if windows or icons are in different places. Some programs, such as Tempo II from Affinity MicroSystems and QuicKeys from CE Software, remember somewhat higher-level commands, but in general, it is necessary to have specific high-level knowledge about the application being run to make transcripting useful [3].

The seminal system that used demonstrational techniques for programming is Pygmalion [15], which supported programming using icons. Industrial robots have long been programmed by example. The trainer of the robot moves the robot's limbs through the desired motions, and the robot records these for later re-use. SmallStar [4] allows the end user to program a prototype version of the Star office workstation [16]. When programming with SmallStar, the user enters program mode, performs the operations that are to be remembered, and then leaves program mode. The operations are executed in the actual user interface of the system, which the user already knows. A textual representation of the actions is generated, which the user can edit to differentiate constants from variables and explicitly add control structures such as loops and conditionals.

Systems with Inferencing

The use of inferencing with demonstration to create Programming-by-Example systems has a long and rather unsuccessful history. For instance, one system [14] tried to generate Lisp programs from examples of input/output pairs, such as (A B C D) ==> (D D C C B B A A). This system is limited to simple list processing programs, and it is clear that systems such as this one are not likely to generate the correct program. In general, induction of complex functions from input/output is intractable [1]. Autoprogrammer [2] is typical of a class of PBE systems that attempt to infer general programs using examples of *traces* of the program execution. The user gives all the steps on one or more passes through the execution of the procedure on sample data. Then, the program tries to determine where loops and conditionals should go, as well as which values should be constants and which should be variables.

The use of inferencing in user interfaces has been more successful when the domain in which inferencing is performed is significantly smaller than general-purpose programming. For example, in "Editing by Example" (EBE), the domain is limited to simple transformations in a text editor [13]. The system compares two or more examples of the inputs and resulting output of a sequence of editing operations in order to deduce what are variables and what are constants. The correct programs usually can be generated given only two or three examples, and there are heuristics to generate programs from single examples. The primary inferencing here is differentiating variables from constants.

There are a number of popular systems that use inferencing in very simple ways. For example, the Macintosh programs Adobe Illustrator and Claris MacDraw remember the transformations used on graphic objects after a "Duplicate" operation and guess that the user wants the same transformations for new objects, so they are applied automatically. In Microsoft Word 4.0, the "Renumber" command will look at the first paragraph in the selection to guess how the number at the beginning of paragraphs should look.

The NOTECH text formatter [5] allows users to type documents in plain text, with no formatting commands, and tries to infer the appropriate formatting from the spacing and contents of the document to produce attractive laser-printer output. For example, a single line is assumed to be header, a group of short pieces of text separated by tabs are assumed to be a table, and text that contains Pascal or C statements is formatted as code. NOTECH is a pre-processor for T_EX and uses a set of rules to parse the input.

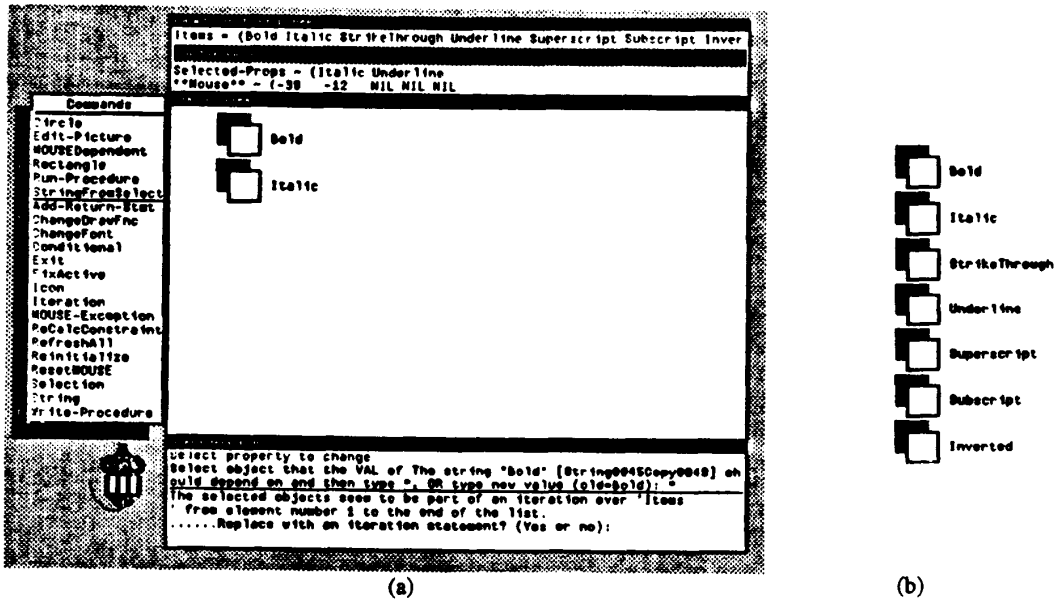

Figure 2: When the user draws the first two elements of a list (a), Peridot infers the need for an iteration (see the prompt window at the bottom). If the user confirms this, the rest of the items of the iteration are created automatically (b). Peridot therefore infers the need for an iteration from two examples.

The Peridot system [7, 11], allows a designer to create user interface components such as menus, scroll bars and buttons with a graphical editor. It successfully uses inferencing in three ways. First, it infers graphical constraints. As the user draws what the interface should look like, the system is always checking to see if there appears to be a relationship between the newly drawn or edited object and others in the picture. Second, Peridot infers iterations and conditionals automatically. For example, if the user creates the first two items from a list of check boxes, the system will create the rest automatically (see Figure 2). The third way is that Peridot infers how the graphics should respond to the mouse. Based on the position of an icon which represents the mouse, the system infers what objects should change and how.

Expanding on the success of Peridot, the Lapidary interface builder [9] allows all application-specific graphical objects to be created by demonstration without programming. For example, the designer can draw examples of the boxes and arrows that will be the nodes and arcs in a graph editor. Lapidary is part of the Garnet system [12].

MetaMouse [6] is also based on a graphical editor, but it watches as the user creates and edits pictures in a 2D click-and-drag drafting package, and will try to generalize from the actions to create a general graphical procedure. If the user appears to be performing the same edits again, the system will perform the rest of them automatically. Inferencing is used to identify geometric constraints in editing operations, to determine where conditionals and loops are appropriate, and to differentiate variables from constants.

New Application Areas

I believe that demonstrational interfaces can be applied to many new application areas in the future. Some that I have thought of include:

1. Allowing arbitrary editing of the pictures on business graphs while still retaining the connection to the data.
2. Customizing the display of data structures in a debugger by editing example displays.
3. Specifying the pictures for scientific visualization without requiring programming.
4. Creating macros with conditionals and loops in direct manipulation interfaces, such as the Macintosh Finder.
5. Creating formatting macros in WYSIWYG text editors.
6. Making it easier to get objects aligned correctly in graphical editors and CAD programs.
7. Generalizing procedures from spreadsheet formulas and macros.
8. Creating educational software.
9. Creating animations.
10. Designing the user interface for software.

We are investigating a number of these areas at CMU. For example, we are developing a new text formatter to investigate number 5 under a grant from Apple Computer. Number 2 may be investigated as part of the MacGnome project [8], which provides pictorial visualizations for Pascal data structures. Number 10 is being studied as part of the Lapidary interface builder for the Garnet user interface development environment [9].

General Paradigms and Models

Since the area of demonstrational interfaces is so new, it is not yet possible to taxonomize the various options and features that can be used. However, it would be quite useful for future projects to have a more formal model of this domain. As a first attempt, we can identify the following ways that demonstration has been used in interfaces. The systems mentioned were described earlier.

- Record a sequence of actions to form a macro that can be used again. This is most useful when some aspects of the macro can be generalized into parameters. Examples: EMACS, SmallStar, MacroMaker, MetaMouse.
- Given examples of input and output, determine a more general program. Examples: I/O Pairs, and Editing by Example.
- Infer repetitive actions so the system can perform the rest of the actions for the user. Examples: Peridot, MetaMouse.
- Given one or more pictures of an object, determine which parts are constant and which should be different. Create a "prototype" object from which new objects can be created. Examples: Peridot, Lapidary.
- Using application-specific semantic knowledge, determine high-level properties from low-level input. Examples: Peridot, NOTECH, MetaMouse.

Further paradigms may be discovered in the future, and the ones listed will be refined.

Research Problems

Although some interfaces that use demonstrational techniques have been built, there has been no systematic study of this technology, and it has not been used in any significant way by a commercial system. Some reasons for this are:

- **There are few existing systems using these techniques, so people are not yet convinced that they are feasible and beneficial.** Future research should create example systems in different application areas and release these systems so they can be used by many people.
- **It is not obvious how to use demonstrational techniques.** Some aspects of the user interface are best performed by menus, some by direct manipulation, and some by demonstration, and it is an interesting challenge to determine which is most appropriate for what.
- **All inferencing systems will sometimes guess wrong.** User interface designers are reticent to use a technology that may make errors. Human factors studies are required to determine when inferencing is beneficial in spite of the occasional errors.
- **It is difficult to provide appropriate feedback.** Because the system can be wrong, it is important for the system to show the users what the system is proposing, so they can verify and correct the inferences. It is not obvious how to provide this feedback. Peridot [7] uses a question-and-answer style to ask the user if each inference was correct. This can be very tedious, and it is modal since the user has to answer the questions before doing other operations. Other systems have simply performed the inferences without providing the user with any feedback or opportunity to undo, and therefore were seldom used. A new style that is user-friendly and compatible with direct manipulation interfaces must be developed.
- **Sometimes Demonstrational interfaces will be *harder* to use,** because:
  - The user may know exactly the name of the relationship that is desired so it might be easy to specify, and it might be more trouble to demonstrate it by example. This can be overcome by providing both specification and demonstrational interfaces to the same operation.
  - When the system guesses incorrectly the user must perform *more* work to detect the error and abort or undo the inference. If the error is undetected by the user, the system will create an erroneous procedure. This problem can be partially overcome by supplying appropriate prompting and feedback along with an "Undo" command.
- **Demonstrational systems are difficult to build.** All existing programs have been separately and laboriously implemented by hand. Toolkits and other support software are needed for demonstrational interfaces.

Conclusions

Demonstrational techniques can substantially improve a wide class of user interfaces and applications. Allowing the system to guess generalizations from the examples adds significantly to the power and ease-of-use of direct manipulation interfaces. These techniques can also make the user interfaces of programs more powerful and exciting without increasing the complexity to the end users. More research is needed, however, to solve the remaining problems and to conclusively show that demonstrational interfaces are viable and easier to use. You can help develop this exciting technology, which may be the next important step beyond the direct manipulation interfaces of today.

Acknowledgements

This research was partially funded by Apple Computer, Inc, and partially by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical

Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Apple Computer, the Defense Advanced Research Projects Agency, or the US Government.

For help with this paper, I want to thank Bernita Myers.

References

1. D. Angluin, and C.H. Smith. "Inductive Inference: Theory and Methods". *Computing Surveys 3*, 15 (Sept. 1983), 237-269.

2. Alan W. Biermann and Ramachandran Krishnaswamy. "Constructing Programs from Example Computations". *IEEE Transactions on Software Engineering SE-2*, 3 (Sept. 1976), 141-153.

3. Richard Joel Cohn. Programmable Command Languages for Window Systems. Tech. Rept. CMU-CS-88-139, Carnegie Mellon University Computer Science Department, June, 1988.

4. Daniel C. Halbert. *Programming by Example*. Ph.D. Th., Computer Science Division, Dept. of EE&CS, University of California, Berkeley, CA, 1981.

5. R.J. Lipton and R. Sedgewick. *NOTECH: Typesetting without Formatting*. Princeton University, 1990.

6. David L. Maulsby and Ian H. Witten. Inducing Procedures in a Direct-Manipulation Environment. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 57-62.

7. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.

8. Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic Data Visualization for Novice Pascal Programmers. 1988 IEEE Workshop on Visual Languages, IEEE Computer Society, Pittsburgh, PA, Oct., 1988, pp. 192-198.

9. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Objects by Demonstration. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 95-104.

10. Brad A. Myers. "Taxonomies of Visual Programming and Program Visualization". *Journal of Visual Languages and Computing 1*, 1 (March 1990), 97-123.

11. Brad A. Myers. "Creating User Interfaces Using Programming-by-Example, Visual Programming, and Constraints". *ACM Transactions on Programming Languages and Systems 12*, 2 (April 1990), 143-177.

12. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer 23* (1990), To appear.

13. Robert P. Nix. "Editing by Example". *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 600-621.

14. David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring Lisp Programs from Examples. Fourth International Joint Conference on Artificial Intelligence, IJCAI'75, Tbilisi, USSR, Sept., 1975, pp. 260-267.

15. David Canfield Smith. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser, Basel, Stuttgart, 1977.

16. David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem. "Designing the Star User Interface". *Byte* (April 1982), 242-282.

17. Richard M. Stallman. Emacs: The Extensible, Customizable, Self-Documenting Display Editor. Tech. Rept. 519, MIT Artificial Intelligence Lab, Aug., 1979.