Demonstrational Interfaces: A Step Beyond Direct Manipulation

Brad A. Myers

August 1990
CMU-CS-90-162

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Direct manipulation interfaces, where objects on the screen can be pointed to and manipulated using a mouse and keyboard, are now almost universally accepted. However, some limitations of these interfaces are well known. These include the lack of programmability and the difficulty of providing abstract commands. *Demonstrational interfaces* can overcome these problems while still providing the benefits of direct manipulation. A "demonstrational interface" watches while the user executes conventional direct manipulation actions, but creates a more general abstraction from the specific example. For instance, the user might drag a file named "v1.ps" to the trash can, and then a file named "v2.ps", and a demonstrational system might automatically create a macro to delete all files that end in ".ps". This paper defines demonstrational interfaces, presents a number of examples, and then discusses some potential application areas.

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.

This is a longer version of the author's invited address to the *IEEE Workshop on Visual Languages*, Chicago, Ill, October 6, 1990, published as "Invisible Programming" in the proceedings.

Keywords: User Interface Styles, Demonstrational Interfaces, Direct Manipulation, Programming-by-Example, Inferencing.

1. Introduction

In his classic 1983 article, Ben Shneiderman introduced the concept of a "direct manipulation" interface [Shneiderman 83]. With the advent of the Apple Macintosh in 1984, this style of interface became more popular, and now is widely accepted. However, there are some well-recognized limitations of conventional direct manipulation interfaces. For example, it is common in Unix for users to write parameterized macros ("shell scripts") that perform common and repetitive tasks, but this is very difficult or impossible in the Macintosh Finder and other "Visual Shells." As a result, experienced users of direct manipulation interfaces find that complex, higher-level tasks that occur commonly are more difficult than should be necessary.

Demonstrational interfaces allow the user to create parameterized procedures and other high-level abstractions without requiring the user to learn a programming language. The key feature of a demonstrational interface is that the user performs actions on concrete example objects (often, by direct manipulation), but a more general-purpose procedure is created. The term "demonstrational" is used because the user is *demonstrating* the desired result using example values.

Demonstrational interfaces can also make direct manipulation interfaces easier to use. I believe that demonstrational interfaces are a significant advance over conventional direct manipulation interfaces, in the same way that direct manipulation is an advance over command line interfaces.

This paper more formally defines demonstrational interfaces and related terms, and then discusses why they are important and how they can be used.

2. 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 infer computer programs from examples of input and output [Shaw 75]. More successful programs limit the inferences to a specific domain. For example, Peridot [Myers 88a] 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 text editor [Stallman 79] 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.

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. Note that it is not sufficient for the interface to be used for *entering* or defining programs, since this would include all text editors. The interface itself must be programmable.

Demonstrational interfaces that provide programming capabilities are called *Example-based Programming* [Myers 90a]. When inferencing is used, these are called *Programming-by-example*, which comes under the topic of "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. Example-based Programming 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.

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 shows how these categories create a taxonomy for classifying systems.

3. Motivation for Demonstrational Interfaces

Since most demonstrational interfaces are also direct manipulation interfaces, they share all of the advantages of that style [Shneiderman 83]. In addition, demonstrational interfaces:

- Can provide programming capabilities to users without requiring any special programming knowledge,
- Can be even easier and more efficient to use than conventional direct manipulation interfaces.

These advantages are discussed in the following sections.

3.1 Programming Capabilities

The vast majority of people who use computers do not know how to write conventional computer programs. As computer usage increases, this will become even more true. However, providing programmability is important to support end user customization. Software supplied by vendors rarely does exactly what the customer requires, so the ability to change the software is quite desirable. Obviously, computers are programmable, but not for the typical user. Spreadsheets are an example of how successful a product can be when it *does* provide programmability. Users write formulas and macros for spreadsheets, which is a form of programming. Unfortunately, for other types of applications, there is no natural way to provide programming capabilities. This is especially true for graphical applications, such as drawing, CAD/CAM, and iconic programs, where there is usually no textual representation of the interactive commands executed with a mouse or other pointing device.

Not Demonstrational

	Not Intelligent	Intelligent
Not Programmable	• *Conventional Direct Manipulation Interfaces*	• *Natural Language Interfaces*
Programmable	• *Unix Shell*	• *Programmer's Apprentice*

Demonstrational

	Not Intelligent	Intelligent
Not Programmable	• *Emacs* • *Macro Makers* • *Industrial Robots*	• *Editing-by-Example* • *NOTECH*
Programmable	• *Pygmalion* • *SmallStar*	• *I/O Pairs* • *Peridot* • *Lapidary* • *MetaMouse*

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

Demonstrational techniques can be used to provide these programming capabilities without requiring the user to learn a programming language. With a demonstrational interface, the user can give the normal commands of the system. In addition to performing the customary action, however, the commands are also 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. The result is a macro or procedure that can be used in multiple contexts. This technique has been successfully demonstrated by a system supporting the desktop metaphor [Halbert 81] and for user interface construction [Myers 88a].

Users will find it easier to create procedures this way than by learning a programming language and writing programs. First of all, it is well known that learning to program is very difficult for most people. Also, most 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 solve specific problems. This helps the students understand the general principles. Fewer errors are made when working out a problem on an example as compared to performing the same operation in the abstract, as in conventional programming [Smith 82]. The user 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 [Smith 77].

3.2 Easier to Use

Demonstrational interfaces can be even easier to use than conventional direct manipulation interfaces, in some cases. 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. However, if the system can infer what the user's intentions are while performing a composite operation, it can save the user from having to perform a number of steps. 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 creating drawings and animations [Maulsby 89].

In addition to helping the user avoid repetitive actions, demonstrational techniques can also be used to infer semantic, high-level, 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 minimum and maximum sizes, and the system would then automatically create the code, as in Peridot [Myers 88a].

Another example of inferencing for semantic properties is the determination of the appropriate "constraints" that should be applied to objects. Constraints are relationships among objects that are maintained by the system. For example, if an architecture system can infer that the windows being placed are on the second floor of a building, then it can automatically insure that they stay between the floor and ceiling of that floor.

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 (such as "make-evenly-spaced-horizontally") that will give the desired result, whereas the system may be able to tell which one 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.

Of course, there are some disadvantages to the demonstrational style. These are discussed in section 9.

4. Survey

The next sections use the taxonomy of Figure 1 to survey various interfaces.

4.1 Not Demonstrational

Systems that are not demonstrational, programmable or intelligent are just the conventional user interfaces, including non-programmable command lines and most direct manipulation systems.

The intelligent, non-demonstrational, non-programmable interfaces include most of the systems developed by AI researchers, including natural language interfaces, adaptive interfaces, etc.

All of the conventional systems that allow the user to program the interface are included in the category for programmable, non-intelligent interfaces. This includes the Unix Shell and the programming language embedded in Emacs (called MockLisp in some versions).

A system that might be classified as intelligent, programmable, and not demonstrational is the Programmer's Apprentice [Rich 88], which uses AI to help the user create programs.

4.2 Demonstrational, Not Programming Systems, Not Intelligent

Perhaps the simplest demonstrational interfaces are keyboard macros for text editors such as Emacs [Stallman 79]. Here, the user goes into program mode, gives a number of commands in the usual way, and then leaves program mode. The commands execute normally while the macro is being created, so the user only has to learn two new commands to create macros: Start-Recording and Stop-Recording, since the rest of the commands are the ones that are used every day while editing. To replay the macro, the user moves the cursor to the appropriate new place and gives one additional command: Execute Macro.[1]

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. For example, the transcript will save information such as "the mouse button went down at location (23,456)". The fact that there was a particular icon at that location will not be recorded. If the windows and icons move after the macro is created, the locations saved may not correspond to the correct objects, so the macros will not work correctly. 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.

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 [Tanner 79].

---

[1]The embedded MockLisp programming language is not used when creating these keyboard macros, so they are not categorized as programmable. Emacs provides many different ways for the user to extend the user interface.

4.3 Demonstrational, Not Programming Systems, Intelligent

In "Editing by Example" (EBE) [Nix 85], demonstrational techniques are used to create transformations of the text (replace *this* with *that*). The system compares two or more examples of the input 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. For instance, the system can deduce a program to convert all occurrences of "@i(<*text*>)" to "{\sl <*text*>}" from editing "@i(O.K.)" to "{\sl O.K.}" and then editing "@i(Boston Post)" to "{\sl Boston Post}". This program can then be run on other parts of the document. Unlike the Emacs macros, however, EBE uses the *results* of the text editing; not the particular editing operations used. 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 (see Figure 2). In Microsoft Word 4.0, the "Renumber" command will look at the first paragraph in the selection and see if there is some form of number there. If so, it will use the formatting of that number to determine how the numbers at the beginning of all the paragraphs should look.

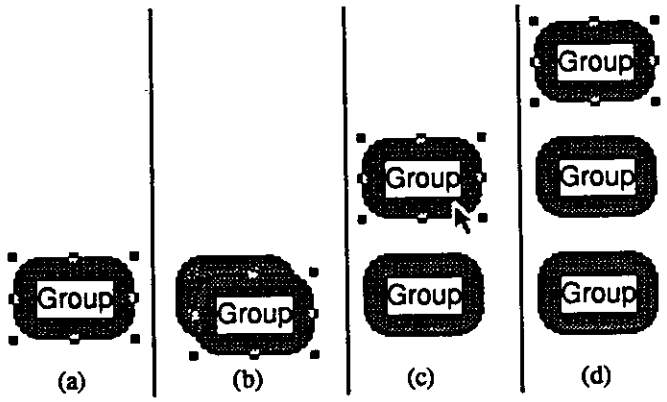


Figure 2: When an object (a) is selected and the "Duplicate" command is executed in MacDraw, the new object is put at a standard offset from the original (b). If the new object is then moved (c) and the duplicate command given on it, then the subsequent object will be offset from the second object the same as the second was from the first (d). This is a simple example of inferring the desired position from the user's example.

Another small use of inferencing from an example is the formatting for the date in the Scribe text formatter [Unilogic 85]. When specifying the way the date should be printed, the user can supply an example in "nearly any notation" for the particular date March 8, 1952, and Scribe will convert this for use with the current date. For example, formats like the following are accepted for specifying how the

current date should be printed:

```
@Style(Date="03/08/52")
@Style(Date="March 8, 1952")
@Style(Date="Saturday, 8th of March")
@Style(Date="8-Mar-52")
@Style(Date="the eighth of March, nineteen fifty-two")
```

The NOTECH text formatter [Lipton 90] 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.

4.4 Demonstrational, Programming Systems, Not Intelligent

All programmable, demonstrational systems are called "Example-Based Programming." When there is no intelligence, they are called "programming-with-example" systems.

The seminal system that used demonstrational techniques for programming is Pygmalion [Smith 77], which supported programming using icons. SmallStar [Halbert 81] allows the end user to program a prototype version of the Star office workstation desktop. The desktop, sometimes called the "Visual Shell" is an iconic interface that supports file creation, deletion, moving, copying, printing, filing in directories (called folders), etc. [Smith 82]. 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.

4.5 Demonstrational, Programming Systems, Intelligent

Demonstrational systems that are programmable and intelligent are called "programming-by-example" systems.

The use of inferencing with demonstration to create programs has a long and rather unsuccessful history. For instance, one system [Shaw 75] 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 [Angluin 83]. Autoprogrammer [Biermann 76] 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 of 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, the Peridot system [Myers 88a] 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. For example, if a rectangle is drawn approximately centered inside a circle, Peridot will ask the user whether it should be adjusted to be exactly centered. The user can confirm the inference or cancel it (in which case other inferences are attempted). By automatically inferring these relationships, Peridot is able to install constraints that allow the relationships to be maintained at run time even if the values change. For example, if the circle changes size or is moved, the rectangle will be adjusted automatically. In addition, the inferencing helps the user by automatically "beautifying" the picture since objects can be drawn quickly and sloppily and the system will neaten them automatically.

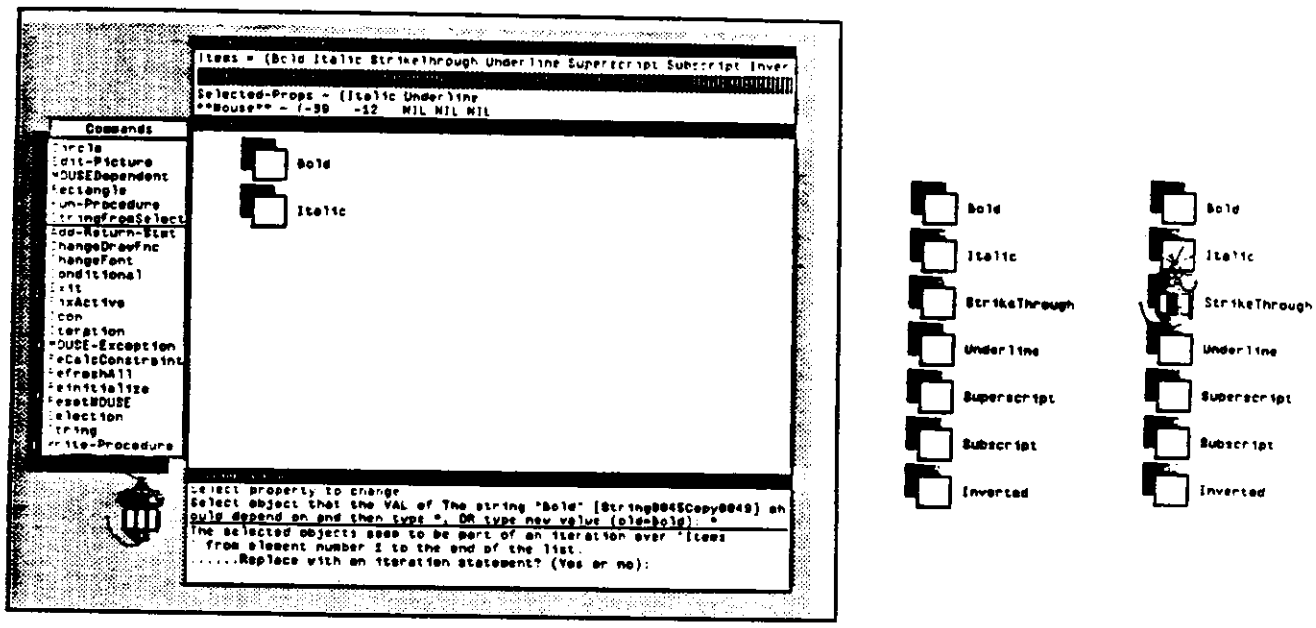
The second way that Peridot uses inferencing is to add iterations and conditionals. For example, if the user creates the first two items from a list of check boxes, the system will create the rest automatically (see Figure 3). The third way is that Peridot infers how the graphics should respond to the mouse. Based on the position of an icon which represents the mouse, the system infers what objects should change and how.

Peridot was tested on five non-programmers and five programmers, and all were able to create user interface elements such as menus after about an hour and a half of instruction. Creating a menu with a custom appearance took them about 15 minutes, whereas it took the most experienced programmers between one and eight hours to code using conventional programming languages [Myers 88a].

Expanding on the success of Peridot, the Lapidary interface builder [Myers 89] 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 [Myers 90b].

Metamouse [Maulsby 89] 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.

Allen Cypher at Apple Computer, Inc. is building a "smart macro" tool, named Eager, for the HyperCard environment. HyperCard is a Macintosh program that allows users to build custom, direct-manipulation applications. Like Metamouse, Eager infers an iterative program to complete a task after the user has performed the first two or three iterations. An important innovation in Eager is its technique for providing feedback to the user about how the system has generalized the user's actions. Eager



(a) (b) (c)

Figure 3: When the user draws the first two elements of a list (a), Peridot infers the need for an iteration (shown in 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. When the mouse icon is placed over the checkmark (c), Peridot infers that it should be a feedback object.

provides feedback by *anticipation*: when it recognizes a pattern, it infers what the user's next action will be, and uses color to highlight the location where the user will next click the mouse (e.g. a menu item, or a button on the screen) or to display the text that the user is expected to type next. If the user performs the anticipated action, this confirms the inference; if the user performs a different action, this provides a new example for Eager to use in determining a better inference about how to generalize the user's actions. The system will also infer iterations if the user performs the same action on multiple cards.

5. Ways to use Demonstrational Techniques

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, we can identify the following ways that demonstration can be used in interfaces:

- Smart recorders,
- Generalizing from input/output,
- Handling repetitive actions
- Neatening pictures,
- Generalizing a picture, and
- Determining high-level properties.

5.1 Smart Recorders

Here, the system records 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. In Emacs, the keyboard macros are implicitly parameterized by the cursor position. When giving the example, the user has the cursor at a particular spot, and all editing operations are relative to that location. After the macro has been defined, the cursor is moved to a new location and the macro is executed there. In SmallStar, the user must explicitly edit a textual representation of the macro to specify which objects should be variables. Metamouse assumes that every value is a variable, and asks the user to confirm or define constraints to determine the value.

5.2 Generalizing from Input/Output

Given examples of input and output, some systems try to determine a more general program. Examples include I/O Pairs and Editing by Example. Typically, without further information, input and output pairs are not sufficient to create a useful program.

5.3 Handling Repetitive Actions

Many demonstrational systems try to determine when the user is performing a similar action to a previous one, to try to form a loop. Then the system can execute the loop to handle the rest of the items. One example is Peridot (Figure 3). Similarly, Metamouse continually watches the user's actions and tries to guess whether a loop can be created. For example, Metamouse can move all the rectangles horizontally in a window until they touch a line after the user has demonstrated moving the first one or two. In SmallStar, the user has to insert loops explicitly, but then the rest of the iterations are executed by the system.

5.4 Neatening Pictures

PED [Pavlidis 85] and Peridot use the inferred graphical relationships to adjust the picture. This allows the user to draw objects sloppily and have the system neaten them up, which can be much faster and more accurate than using grids. Examples of the inferred relations are that two lines should be perpendicular, or that one rectangle should be exactly centered within another.

These relationships might also be retained after the neatening. This will help the user maintain the correct picture while it is being edited. In this case, the system is actually establishing graphical constraints. Peridot and Metamouse do this, but PED does not.

Inferencing could also be used to help *create* the objects neatly. In many direct manipulation interfaces, certain points attract the cursor when it is moved near them, which makes it easier to create the desired pictures. For example, if the cursor is close to the end point of a line, it might jump to be exactly on the end point (as if attracted by *gravity*). This idea can be extended to provide inferred gravity points. For example, if the user has drawn two objects a certain distance apart, the system might insert a gravity point at the same distance from the second object, in case a third is desired. In a similar way, the positions and sizes for alignment lines and circles, as used in computerized "ruler" and "compass" tools

such as the Snap-Dragging system [Bier 86], might also be inferred from the drawing.

5.5 Generalizing a Picture

In Peridot and Lapidary, the user draws an example of an object that will be used in the interface, and the system generalizes to create a "prototype" object, from which "instances" can be created at run-time. For example, when designing a menu, the prototype will be created using a specific example set of strings, but the instances will use different strings. Typical properties of the objects that change in instances include the position and size, and the label strings. When generalizing the pictures, these systems must determine which properties of the objects should be constant and which change.

5.6 Determining High-Level Properties

Since many demonstrational interfaces are limited to a particular domain, they can use domain-specific knowledge to generalize from the low-level actions and objects that the user creates to establish higher-level properties. For example, Peridot can infer that a check mark is serving as a feedback object, and so should follow the mouse (Figure 3-c). NOTECH uses this technique extensively to determine what role the parts of the text play in the document.

6. Feedback in Demonstrational Interfaces

Any inferencing system will sometimes guess wrong, so an important problem is how to inform the user of the inferences made by the system. The tradeoff is between giving the users confidence that they know what the system is doing and that it is correct, versus the interruption and loss of efficiency that feedback often causes. The next sections discuss some of the different ways that feedback has been provided in demonstrational systems.

6.1 Showing the Generated Code

In some early systems, such as I/O Pairs and Biermann's system using Traces, the only way to determine what was inferred was to look at the generated code. This essentially defeats the purpose of using a demonstrational interface, however. Some modern systems, such as SmallStar, also display the code, either to allow editing or to help teach the user the underlying programming language, when that is appropriate.

6.2 Question-and-Answer

In the question-and-answer style, which is used in Peridot and Metamouse, the system asks the user, using canned English sentences, if each inference is correct. This is easy to implement and was well-liked by Peridot's users, but it can be modal, disruptive, and error-prone. It can either be implemented using a special prompt window or using pop-up alert boxes, depending on how often the questions arise.

6.3 Dialog Boxes

Some systems show the user what was guessed by setting values in menus or dialog boxes (also called property sheets or forms). This style is used by Eager and Lapidary. The user can look at the dialog box to determine what inference was made, and then modify it if necessary. For example, if the user is designing the formatting for an itemized list, the system might create a dialog box showing all the possibilities for the formatting of the various parts with the inferred choices highlighted. If the user is happy, then the "OK" button can be hit. If not, then different values could be selected. In addition, there may be a "Next Guess" button to have the system try a different guess if the first try was incorrect.

6.4 Highlighting of Significant Features

A system might show which properties of the example are considered important by using special highlighting. For example, Metamouse puts different styles of selection boxes (sometimes called "handles") on graphical objects to show which locations are considered important (see Figure 4). The user can click on the boxes to change their status if the system has guessed incorrectly.

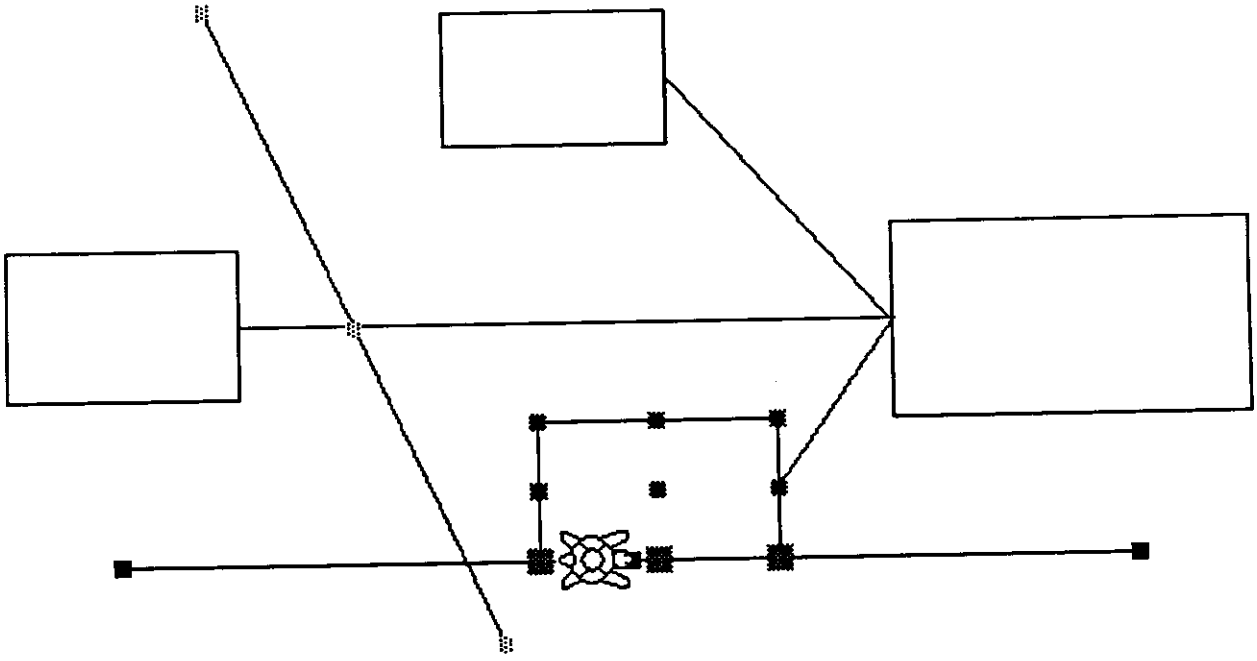


Figure 4: Metamouse shows the selection "handles" differently depending on the role that it has inferred for the object. Black handles indicate the object being grasped, dark grey handles are used for an object directly touched by the turtle icon, and light grey is for indirect touches. The size of the handles represent the importance. [Picture courtesy of David Maulsby.]

6.5 No Feedback

Another possibility is to just go ahead and perform the inferred action immediately, and hope the action itself will provide sufficient feedback. Obviously, this is most successful where the action will have some visible consequence, such as drawing or editing some graphics. It is especially important here to have an easy-to-use "Undo" facility.

7. Software Architectures

It is relatively difficult to implement demonstrational interfaces. Unfortunately, since the area is so young, there are no well-recognized ways for organizing the software, and there are certainly no toolkits to help generate demonstrational interfaces. All existing systems have been individually crafted. However, we can identify some common components.

Most demonstrational systems with inferencing have used a rule-based mechanism to guess the generalizations. The "condition" of the rule determines if the rule should be applied in the current context, and if so, then the "action" fires to assert the generalization. In some systems, rules also contain messages to serve as feedback that the rule is going to fire. As an example, a set of rules to guess how a feedback object should move with the mouse might include:

1. If the feedback object is on top of an element of a list, then assume that it moves from item to item, as in a menu.
2. If the feedback object is constrained to be inside of a narrow rectangle, then assume that it is an indicator in a scroll bar or slider, so it can only move in one dimension and must stay inside the bounding rectangle.

Typically, the rule systems used in demonstrational interfaces are much simpler than those in AI expert systems, and are often custom built. One reason the more complicated "expert system shells" are not needed is that most demonstrational systems have only a small number of rules. For example, Peridot uses only 50 rules to infer graphical relationships and 6 to infer how feedback objects follow the mouse. Metamouse uses 13 rules to identify the most likely constraints on an action.

In the future, we can expect that toolkits will be developed to help create demonstrational interfaces. These might contain:

- Rule mechanisms, including appropriate search algorithms for when multiple rules apply.
- A set of rules that are commonly used (such as finding loops or similarities in objects).
- Ways to represent the user interface objects so that the rules can easily determine the necessary information.
- Various feedback mechanisms integrated with the rules to show the user what is being inferred.

8. New Application Areas

As may be evident from the survey section, there are as yet no large-scale uses of demonstrational techniques in commercial systems. However, I believe that demonstrational interfaces can be successfully used in many different kinds of programs. A demonstrational interface might be appropriate for an application area when one or more of the following holds:

- There is high-level knowledge of the domain that could be represented in the program,
- In some situations, the user repeatedly performs the same low-level commands.
- The application has both a textual, command-line interface and a graphical, direct manipulation interface, and there are some programming-like features available in the former but missing from the latter.
- Users need to customize the user interface of the program or its output and generalize from the limited options provided.

The following sections discuss some of the areas that I think would benefit from using this technology.

8.1 Macros for Direct Manipulation Interfaces

By adding inferencing to macro recorders for direct manipulation interfaces, like the Macintosh Finder, the system could probably guess what should be variables and constants, and what control structures are needed, and thereby create more useful macros. For instance, if the user dragged a file named "v1.ps" to the trash can, and then a file named "v2.ps", a demonstrational system might automatically create a macro that will delete all files that end with ".ps".

8.2 Business Graphics

Many business graphics packages allow the user to select among a set of graph and chart types, which are then used to display the user's data. Demonstrational techniques could be used to allow the user to customize how the particular display looks. For example, if a bar graph is chosen, but the user wants to change the shape of the bars (e.g., to be people, fish, or oil barrels as in *USA Today*-style charts), one of the bars could be edited. The system would generalize from these edits to make all the bars look the same—there would be no need to individually edit each bar. This could happen in real-time, so that all the bars are updated as the user edits one. Also, if the data subsequently changed, the new bars would still change size automatically, unlike what would happen if the bars had been copied to a conventional graphics program and then edited.

8.3 Data Visualization

Another application area is Data Visualization [Myers 90a], which is where graphics are used to make data in computer programs more understandable. Demonstrational techniques could be used to make it easy for the user to create custom displays. In the MacGnome system, which is a syntax-directed editor for Pascal on the Macintosh, the user can point to any variable and get a picture of the data value in the "standard" pictorial formats (records and arrays use stacked boxes, and pointers use a line with an arrowhead) [Myers 88b]. If a demonstrational editor was added, the user could draw the desired display

using an example of the output data, and the system would apply this to all future displays, even when the data changed.

8.4 Scientific Visualization

A similar technique might be used for scientific visualization. It may be possible to use demonstrational techniques to allow the scientist to demonstrate the desired display on a small sample of data, and have the system generalize the display automatically to the full data set without requiring the assistance of a programmer.

8.5 Drawing Packages

One common problem with simple drawing packages like Macintosh MacDraw is that there is no mechanism for specifying how the graphical objects relate to one another. For example, a line may be drawn to a box, but it will not stay attached if the box is moved. Other drawing packages, including Sketchpad [Sutherland 63] and Juno [Nelson 85], have addressed this problem by allowing the user to specify constraints on the objects. Unfortunately, users often have a difficult time correctly specifying and understanding constraints. Peridot demonstrated that it is possible for the system to successfully guess graphical constraints on objects if the type of drawing is known to the system. This technique could be used in more general drawing packages so that parts of the pictures could be edited and still retain important properties. Another advantage to the demonstrational approach is that the constraints are guaranteed to correctly create the example picture, whereas when constraints are explicitly applied, the user can accidentally set up a situation where there is no possible solution to the constraints.

8.6 Text Formatting

With old-fashioned text formatting languages such as troff, Scribe, and T_EX, users can create whatever appearance is desired by writing macros in the formatter's programming language. These can take parameters, so, for example, a macro can take the chapter title and produce Figure 5. The advantage of this technique is that all chapters are guaranteed to look the same, and that the appearance of all chapter headings can be changed *at the same time* by editing the master definition. The disadvantage, of course, is that complex programming is required.

2

Chapter II

Related Work

Figure 5: A chapter heading that includes different formatting for different parts.

With the advent of direct manipulation, WYSIWYG ("What you see is what you get") editors, much

of this capability has been lost. The "Styles" mechanism supplied by some editors, such as Microsoft Word, is not sufficient, since it can only handle one set of formatting, and cannot be used to insert constant text into the document, such as the word "Chapter" in Figure 5. Using demonstrational techniques, the user could specify the appearance of document portions *by example*, and the editor could retain the direct WYSIWYG style, while still having the advantages of the macro technique. This might be provided by simply allowing the user to select a section of text and tell the system to generate a macro from it. The system could probably infer the structure using some simple heuristics based on the typical structure of the document (numbers versus regular text, special words such as "chapter" and "section," separators such as periods and colons, etc.). The next time a chapter or section was needed, the user could just type "Chapter III: Overview" (or maybe just type "Overview" and apply the newly created macro to it), and the system would immediately format the chapter heading appropriately.

These kinds of macros by example could also be used to specify the formatting of many parts of the document, including page headings and footings, bulleted or numbered lists, tables, bibliographic references, entries in an index, etc.

8.7 User Interfaces for Simulations

When creating a user interface for a computer program that will monitor a simulation or an actual working process, it would be nice if the user could simply *demonstrate* how components of the interface should react to changes in the incoming data. For example, it should be easy to demonstrate how a gauge should change based on a temperature measurement. Similarly, if an item moves along a conveyor belt in a factory, it should be possible to demonstrate how its icon moves on the screen, assuming the movement is generally at a constant speed.

8.8 User Interface Development Environments

Most of the graphics and behaviors of user interfaces (the "look and feel") can probably be specified by demonstration. The Peridot system demonstrated that low level components of user interface can be constructed by example without programming, and our current work on Lapidary extends this idea to other parts of the user interface.

8.9 Computer Game Construction

A popular style of video game has a number of characters, some good and some bad, and they interact in various ways. For example, they jump, walk, shoot, and bump into other characters. Demonstrational techniques could be used to create a "video game construction kit" that would allow the characters to be drawn with a drawing package, and then have their movements and behaviors defined by demonstration. This is feasible since the behaviors and interactions among characters usually are fairly simple.

8.10 Others

There are many other possible applications of this technology, including generalizing procedures from spreadsheet formulas, creation of educational software, creating animations, etc.

9. Research Problems

Although some interfaces that use demonstrational techniques have been built, there has been no systematic study of this technology. A number of significant problems remain to be solved by future research. These include:

- **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. This will help identify what applications are most appropriate for demonstrational interfaces. At CMU, we are investigating demonstrational interfaces for macros for direct manipulation interfaces, data visualization [Myers 88b], text formatting, and user interface development environments [Myers 90b].
- **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 to users in spite of the occasional errors.
- **The inferred procedures may contain errors.** Even if the procedure operates correctly on the example data, it may not work for some other important cases. Of course, studies have shown that user-generated procedures for spreadsheets are often incorrect [Brown 87], so it will be interesting to see whether procedures that are demonstrated or programmed by the end users are more reliable in practice.
- **It is difficult to provide appropriate feedback.** Because the system can guess 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, and when each of the various styles is appropriate.
- **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 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.

10. Conclusions

Demonstrational techniques can substantially improve a wide class of user interfaces and applications. Allowing the system to guess generalizations from the actions of the user 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 user. More research is needed, however, to solve the remaining problems and to conclusively show that demonstrational interfaces are viable and easy to use. Nevertheless, I believe that this exciting technology will be the next important step beyond the direct manipulation interfaces of today.

Acknowledgements

For help with this paper, I want to thank David Maulsby, Dario Giuse, and Bernita Myers.

References

[Angluin 83] D. Angluin, and C.H. Smith.
Inductive Inference: Theory and Methods.
Computing Surveys 3(15):237-269, September, 1983.

[Bier 86] Eric Allan Bier and Maureen C. Stone.
Snap-Dragging.
In *Computer Graphics*, pages 233-240. Proceedings SIGGRAPH'86, Dallas, Texas, August, 1986.

[Biermann 76] Alan W. Biermann and Ramachandran Krishnaswamy.
Constructing Programs from Example Computations.
IEEE Transactions on Software Engineering SE-2(3):141-153, September, 1976.

[Brown 87] P.S. Brown and J.D. Gould.
An Experimental Study of People Creating Spreadsheets.
ACM Transactions on Office Information Systems 5(3):258-272, July, 1987.

[Halbert 81] Daniel C. Halbert.
Programming by Example.
PhD thesis, Computer Science Division, Dept. of EE&CS, University of California, 1981.

[Lipton 90] R.J. Lipton and R. Sedgewick.
NOTECH: Typesetting without Formatting
Princeton University, 1990.

[Maulsby 89] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz.
Metamouse: Specifying Graphical Procedures by Example.
In *Computer Graphics*, pages 127-136. Proceedings SIGGRAPH'89, Boston, MA, July, 1989.

[Myers 88a] Brad A. Myers.
Creating User Interfaces by Demonstration.
Academic Press, Boston, 1988.

[Myers 88b] Brad A. Myers, Ravinder Chandhok, and Atul Sareen.
Automatic Data Visualization for Novice Pascal Programmers.
In *1988 IEEE Workshop on Visual Languages*, pages 192-198. IEEE Computer Society, Pittsburgh, PA, October, 1988.

[Myers 89] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Objects by Demonstration.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95-104. Williamsburg, VA, November, 1989.

[Myers 90a] Brad A. Myers.
Taxonomies of Visual Programming and Program Visualization.
Journal of Visual Languages and Computing 1(1):97-123, March, 1990.

[Myers 90b] 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(11):To appear, November, 1990.

[Nelson 85] Greg Nelson.
Juno, a Constraint-Based Graphics System.
In *Computer Graphics*, pages 235-243. Proceedings SIGGRAPH'85, San Francisco, CA, July, 1985.

[Nix 85] Robert P. Nix.
Editing by Example.
ACM Transactions on Programming Languages and Systems 7(4):600-621, October, 1985.

[Pavlidis 85] Theo Pavlidis and Christopher J. Van Wyk.
An Automatic Beautifier for Drawings and Illustrations.
In *Computer Graphics*, pages 225-234. Proceedings SIGGRAPH'85, San Francisco, CA, July, 1985.

[Rich 88] Charles Rich and Richard C. Waters.
The Programmer's Apprentice: A Research Overview.
IEEE Computer 21(11):11-25, November, 1988.

[Shaw 75] David E. Shaw, William R. Swartout, and C. Cordell Green.
Inferring Lisp Programs from Examples.
In *Fourth International Joint Conference on Artificial Intelligence*, pages 260-267. IJCAI'75, Tbilisi, USSR, September, 1975.

[Shneiderman 83] Ben Shneiderman.
Direct Manipulation: A Step Beyond Programming Languages.
IEEE Computer 16(8):57-69, August, 1983.

[Smith 77] David Canfield Smith.
Pygmalion: A Computer Program to Model and Stimulate Creative Thought.
Birkhauser, Basel, Stuttgart, 1977.

[Smith 82] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem.
Designing the Star User Interface.
Byte 7(4):242-282, April, 1982.

[Stallman 79] Richard M. Stallman.
Emacs: The Extensible, Customizable, Self-Documenting Display Editor.
Technical Report 519, MIT Artificial Intelligence Lab, August, 1979.

[Sutherland 63] Ivan E. Sutherland.
SketchPad: A Man-Machine Graphical Communication System.
In *AFIPS Spring Joint Computer Conference,* pages 329-346. 1963.

[Tanner 79] William R. Tanner (editor).
Industrial Robots - Volume 1: Fundamentals.
Society of Manufacturing Engineers, Dearborn, Michigan, 1979.

[Unilogic 85] *Scribe Document Production Software User*s Manual*
Unilogic, Ltd, Pittsburgh, PA, 1985.