Self-disclosing design tools: A gentle introduction to end-user programming

Chris DiGiano and Mike Eisenberg

Department of Computer Science
University of Colorado at Boulder, CB 430
Boulder, CO 80309-0430
{digi,duck}@cs.colorado.edu

KEYWORDS: end-user programming, learning

ABSTRACT

Programmable tools for design offer users an expressive new medium for their work, but becoming acquainted with the tool's language can be a daunting task. To address this problem, we present a framework for the design of *self-disclosing* tools which provide incremental, situated language learning opportunities for designers in the context of authentic activity. By way of example, we present *Chart 'n' Art*, a programmable application for the creation of graphs and information displays. Chart 'n' Art employs a wide variety of self-disclosure techniques whose purpose is to introduce users to the system's "domain-enriched" dialect of Lisp.

INTRODUCTION

In the 1950's and 60's, literary critic and educator I.A. Richards published a series of books called *Language through Pictures* (Richards, 1973) as a teaching tool for second language learners. Each page of Richards' book consists of a picture and one or more sentences describing the scene in the language to be learned. By following the sequence of pictures and sentences from simple to more complex situations, the reader is supposed to acquire a basic understanding of the language. Richards' pedagogical approach is compelling in that it enables learners to teach themselves a language at their own pace simply by observing connections between images and symbols. His work raises an especially interesting—even urgent—question for the area of computer science education: Can similar approaches be found to support the acquisition of programming languages? This paper outlines one possible method of introducing programming concepts that not only supports self-paced learning as in *Language through Pictures*, but also situates the learning experience in authentic activity.

What kind of activity? Our research focuses on the complex and creative process of design, since it is designers who can clearly benefit from the ability to program their tools. Programming offers designers the opportunity to transcend the built-in functionality of their software, empowering them to be more creative and expressive users. Tools that combine a direct manipulation interface with a domain-oriented language such as the drawing program SchemePaint (Eisenberg, 1995) and the multimedia authoring package Director[1] have been dubbed programmable applications. A major challenge with programmable applications is informing designers of the utility of programming and supporting them in their pursuit of programming expertise. These issues are central to the eventual acceptance and creative use of any application-oriented language, and are in fact the key problems for the entire field of end-user programming (DiGiano and Eisenberg, 1995).

Although end-user programmable systems represent a burgeoning class of software[2], support for those users interested in becoming acquainted with their tool's language is limited. Few organizations formally support the social channels by which experienced users can communicate the cost and benefits of programming to colleagues (Gantt and Nardi, 1992; Nardi and Miller, 1991). Furthermore, the domain specificity and granularity of many embedded languages such as Emacs Lisp (Stallman, 1981) are inappropriate for beginning users (Nardi, 1993, p. 52). With the exception of spreadsheet formulas, most end-user languages fail Nardi's approachability test which says users should be able to readily employ a language after only limited exposure.[3]

Printed tutorials, on-line tutoring programs,[4] and training classes are some of the few support mechanisms widely available to users learning programmable tools. These resources typically have three major drawbacks: 1) they require a significant time investment, 2) they expect the learner to process a large amount of information at once, and 3) they expect the learner to be able map the topics covered to his or her particular tasks. Because of the time and effort required on the part of the user, tutorials and training

[1]Director and Lingo are a registered trademark of Macromedia Corporation.

[2]Microsoft, for instance, has begun integrating its Visual BASIC language into most of its personal productivity software including Word and Excel. (Microsoft, Visual BASIC, Word, and Excel are registered trademarks of Microsoft Corporation.)

[3]As a general heuristic, Nardi suggests that "end-user programming systems should allow users to solve simple problems within their domain of interest *within a few hours of use.*" (italics in original) (Nardi, 1993, p. 45)

[4]Experimental intelligent tutoring systems such as the Lisp Tutor (Anderson, 1985) could hardly be called "widely available," but they do nonetheless suffer from some of the same problems as traditional on-line tutorials.

classes are usually only a last resort (Gantt and Nardi, 1992; Nardi and Miller, 1991).

One possible response to the deficiencies of traditional resources for beginning end-user programmers is to embed contextualized programming language learning opportunities into design tools themselves. Examples of such a mechanism include context-sensitive on-line help such as found in Director for its Lingo language and critiquing systems such as the Lisp Critic (Fischer, 1987). However, both approaches assume the user has a basic knowledge of programming: context-sensitive help systems typically provide information on selected keywords from the language (i.e. system- rather than task-indexed), and critics react to programs already being constructed.

SELF-DISCLOSURE

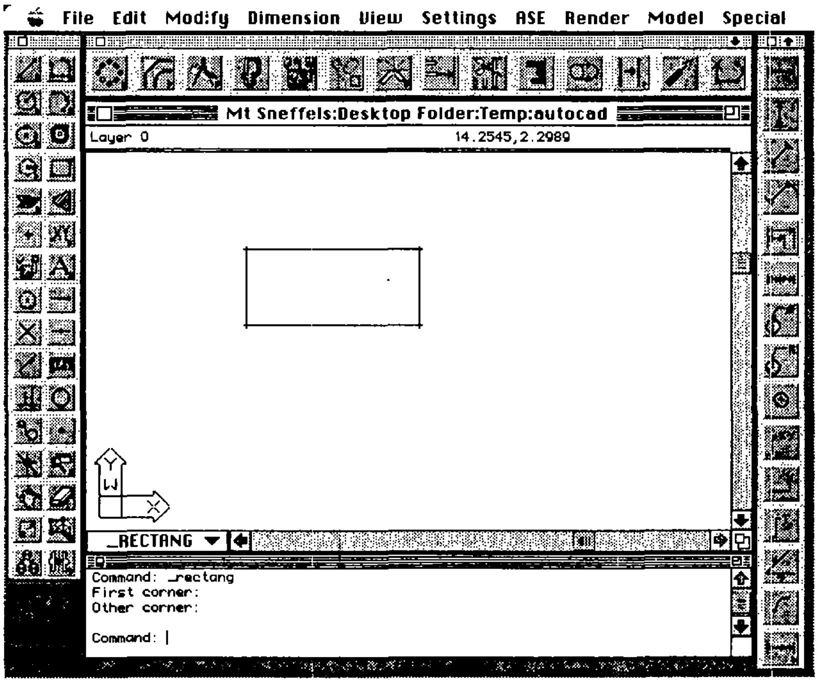
Experience with current resources for presenting end-user languages suggests several desiderata for effective support mechanisms: namely, such mechanisms must 1) reduce both time and effort required by the user, 2) facilitate estimation of costs and benefits, 3) minimize prerequisite programming knowledge, and 4) situate learning in authentic use. In this paper we describe an approach which—like typical context-sensitive help—embeds short, situated learning opportunities into the design tool; but unlike such help systems, this approach has no programming knowledge prerequisites. The technique involves using what has been termed "self-disclosure" to gently introduce designers to programming: that is, the programmable tool "discloses" elements of its language to designers as they use it. By way of example, we present *Chart 'n' Art,* a working prototype of a programmable charting application. Chart 'n' Art illustrates a variety of techniques for self-disclosure as a means of introducing its users to the system's "enriched" dialect of Lisp.

Before presenting Chart 'n' Art itself, we first illustrate the effective use of self-disclosure through an anecdote of a professional designer learning to program; we then characterize both the type of learning that took place and the essential system behavior which enabled that learning. Finally, by way of introducing Chart 'n' Art, we present guidelines for tuning self-disclosure to best fit the needs of designers learning programming for the first time.

LEARNING AN END-USER LANGUAGE: A CASE STUDY

An example of a self-disclosing programmable tool is the computer aided design system, AutoCAD,[5] shown in Figure 1, which has evolved over the years from a simple command line-based DOS program to a mouse-driven multiplatform behemoth. Although users can now interact with the interface by points and clicks alone, the system still makes its command line available in a text-only "Command" window for backwards compatibility, the entering of exact values, and the execution of library functions. Users can also enter coordinate parameters either by typing their values or by using the mouse to select screen locations. What makes AutoCAD self-disclosing is that mouse clicks on one of its extensive toolbars causes the system to disclose the equivalent historical command name in the Command window.

AutoCAD disclosures provide designers with a convenient way to learn its command language. We once interviewed an experienced architect and graphic designer who had started using AutoCAD to calculate the perimeter and area of a floor plan. Her initial approach involved selecting parts of the floor plan with the mouse, applying a menu command to find their points of intersection, and finally using another menu command to compute perimeter and area from these points. When she began with AutoCAD, she had little, if any, programming experience and certainly no knowledge of the


File Edit Modify Dimension View Settings ASE Render Model Special
Mt Sneffels:Desktop Folder:Temp:autocad
Layer 0
14.2545,2.2989
_RECTANG

```
Command: _rectang
First corner:
Other corner:

Command: |
```

Figure 1. AutoCAD's self-disclosing interface. Drawing a rectangle with the mouse reveals the command _rectangle in the bottom Command window.

system's command language. Still, as she started using the tool's mouse to click on her drawing and pull down menu commands she noticed a pattern: each mouse action was followed by text appearing in a window beneath her drawing. She soon realized these were commands she could use to automate her task. It was not long before she learned from AutoCAD's printed manuals how to compose the textual commands into an executable file. Designer had become programmer.

CHARACTERISTICS OF SELF-DISCLOSING SYSTEMS

Psychologists and linguists might describe the kind of learning which took place in the AutoCAD anecdote as *learning by observation* or *incidental learning.* The basic theory of learning by observation is that people notice patterns in the world from which they can incrementally make useful generalizations (DeJong, 1983; Gleitman, 1994). In the

[5]AutoCAD is a registered trademark of Autodesk Corporation.

AutoCAD anecdote the designer observed that direct manipulation actions caused their equivalent linguistic forms to appear in the Command window and from this began acquiring knowledge about the semantics of specific AutoCAD commands as well as the general organization of the language. Users of many spreadsheet programs such as Excel can learn their system's formula languages in a similar way. For example, selecting a column of spreadsheet cells and clicking the mouse over a summation icon generates an expression such as `=sum(A1:A10)` in the last cell of the column, thus revealing part of the formula language to the user.

The following three properties characterize self-disclosing programmable applications which enable language learning by observation:

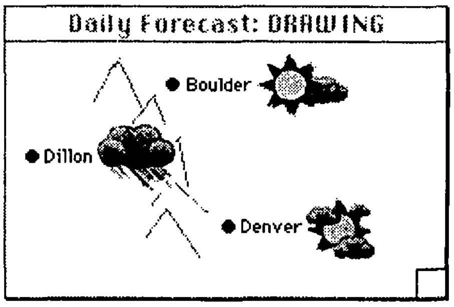
1. For every elementary mouse action which has a command language analog, the system will disclose that expression to the user.
2. The system will indicate groups of disclosed expressions connected with a single operation.
3. Had the user entered the most recently disclosed group of expressions instead of specifying the operation through direct manipulation, the results would have been identical.

Consider how AutoCAD fits each property. 1) Mouse actions for selecting tools, drawing figures, indicating locations, etc. are translated by AutoCAD into the corresponding command language expressions which appear in the system's Command window. 2) Initiating an operation causes a command name to appear after the " Command:" prompt. Subsequent mouse actions then specify the parameters named in the Command window. Completion of the operation is indicated by a new Command: prompt. 3) If users undo their last mouse operation and type the disclosed expressions which were generated in the Command window, the effect is a redo.

TUNING SELF-DISCLOSURE FOR LEARNING

Simply by ensuring that a system exhibits the three self-disclosure properties above, developers can make important progress towards addressing the issues involved in introducing an end-user language. For example, self-disclosing systems of this nature provide a partial solution to the "production paradox" (Carroll and Rosson, 1987), since learning from disclosures can take place after every mouse action *while the designer is designing*. Such self-disclosing systems also expose designers to some of the possible uses of programming: at the very minimum, programming expressions used to imitate direct manipulation actions. Finally, the use of self-disclosure, as specified by the above properties, naturally situates learning opportunities in the context of authentic design activity. That is, the designer can learn about programming expressions directly related to the operations they are currently invoking with the mouse.

Although the three-step recipe for self-disclosure has inherent pedagogical strengths, systems that employ disclosure in this simple way can offer only limited learning opportunities.[6] For example, although AutoCAD fits each self-disclosure property, in order to learn about more complex programming concepts such as expression nesting, iteration, conditionals, or composition designers must interrupt their tasks to read the command language manuals. We believe the technique can be refined to better address the needs of designers learning their tool's application-oriented language. The goal of our research is to provide guidelines for programmable application developers which suggest how to tune self-disclosure so as to offer a gentle, but thorough introduction to the fundamentals of end-user programming.

a.


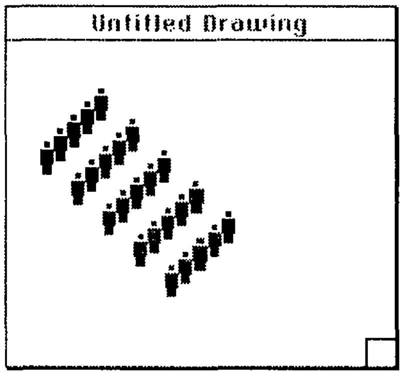
b.


Figure 2. Non-standard diagrams produced by CNA.

CHART 'N' ART: A SELF-DISCLOSING DESIGN TOOL

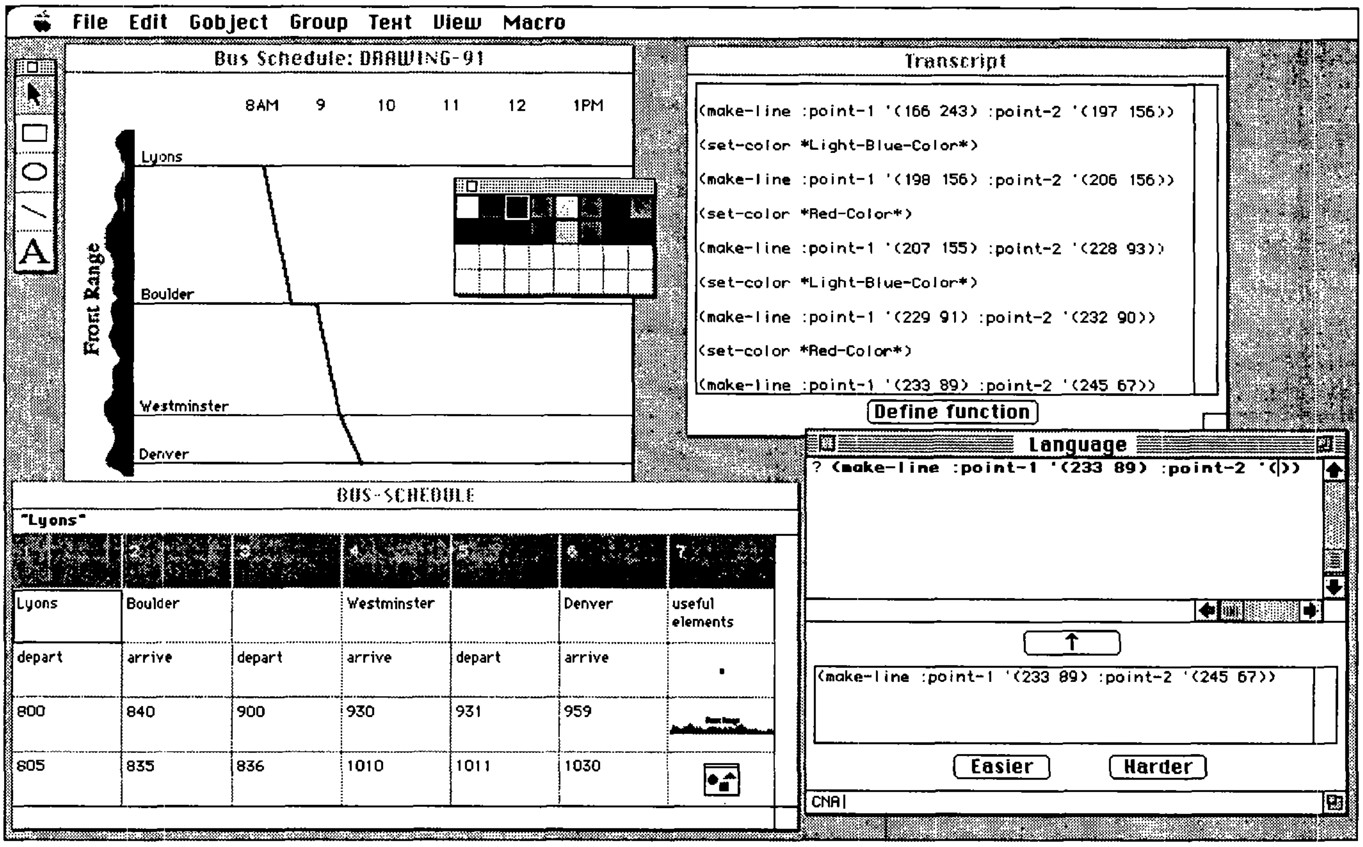
Chart 'n' Art (CNA) is a programmable information graphics tool we have begun developing in order to test and refine guidelines for the use of self-disclosure. CNA supports designers in creating non-standard diagrams such as those depicted in Figure 2 which could not easily be made with commercial packages such as DeltaGraph Pro.[7] Figure 2a is a weather map which was generated directly from forecast data. A short function was written so that a new forecast could be translated to an updated map at the touch of key. Figure 2b is a reconstruction of a unique time line featured in

[6]It is doubtful that developers of most existing self-disclosing tools such as AutoCAD and Excel specifically provide disclosures to help users learn to program them. More likely, self-disclosure is employed to inform users of direct manipulation shortcuts to already known language constructs (e.g. the summation icon as a short cut for the sum expression), or, in the case of AutoCAD, to help wean traditional users off a command line interface.

[7]DeltaGraph is a registered trademark of DeltaPoint Corporation.

Diagram Graphics (Nishioka, 1992) depicting the acceleration of information technology. Each figure represent a human generation, its color the level of information technology available for that generation. Again, a short function was written, this time to translate essentially a diagram "key" into a time line.

Like many charting programs such as DeltaGraph, CNA combines the functionality of a spreadsheet and drawing package. The spreadsheet is used to maintain a "kit" of data, graphical objects ("gobjects"), and functions related to some particular diagram or diagram type. The drawing package is used to construct the actual information displays for the data in the kits. Designers can employ direct manipulation to perform standard drawing and spreadsheet operations. As graphical objects into a drawing, or adjust the attributes of existing graphical elements. Each of the direct manipulation tools mentioned above has a corresponding gobject-maker function such as `make-rectangle` or `make-oval`. Furthermore, CNA programming expressions can be used as a computational "adhesive" that connects the (typically disjoint) worlds of spreadsheet and drawing program by mapping spreadsheet data into graphical elements and vice versa. An example of such a language construct is `copy-gobject` which designers can use to duplicate a kit element and introduce it into a drawing. Since many CNA expressions affect selected gobjects in a window, there is considerable language support for manipulating selections such as `select-up`, `select-down`, and `select-at`.



```
(make-line :point-1 '(166 243) :point-2 '(197 156))
(set-color *Light-Blue-Color*)
(make-line :point-1 '(198 156) :point-2 '(206 156))
(set-color *Red-Color*)
(make-line :point-1 '(207 155) :point-2 '(228 93))
(set-color *Light-Blue-Color*)
(make-line :point-1 '(229 91) :point-2 '(232 90))
(set-color *Red-Color*)
(make-line :point-1 '(233 89) :point-2 '(245 67))
```

```
? (make-line :point-1 '(233 89) :point-2 '()
```

```
(make-line :point-1 '(233 89) :point-2 '(245 67))
```

1	2	3	4	5	6	7
Lyons	Boulder		Westminster		Denver	useful elements
depart	arrive	depart	arrive	depart	arrive	•
800	840	900	930	931	959	
805	835	836	1010	1011	1030	

Figure 3. Chart 'n' Art main windows and palettes. The Bus-Schedule window is an instance of a kit which can hold data, pictures, and procedures for creating the diagrams in a drawing window (top left). The Language window displays the most recent disclosure and allows users to edit and evaluate expression. The Transcript window maintains a history of disclosures.

shown in Figure 3, designers can use a tool palette and color palette to create colored rectangles, ovals, lines, and text which are movable and resizable with the mouse. Users can also employ the mouse to select spreadsheet cells. Menu commands are also considered part of the direct manipulation interface. CNA menus currently offers functions for pasting color pictures into drawings or kits and for aligning graphical objects.

The CNA application-oriented language is an extension of Lisp designed to support the kind of sophisticated data-to-picture translations shown in Figure 2. The language can be used to create or modify spreadsheet cells, introduce new

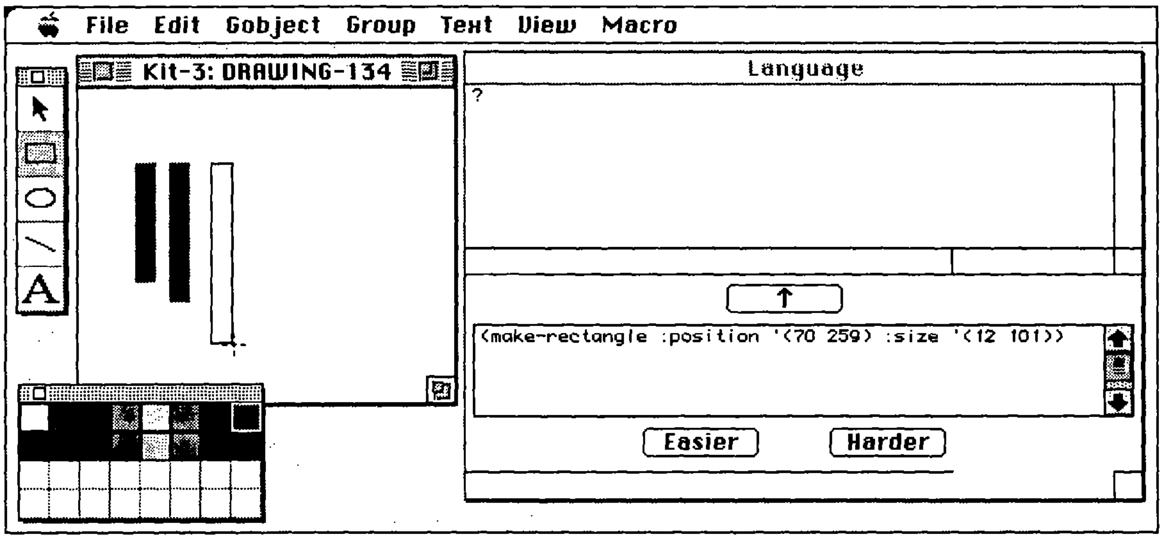
Chart 'n' Art employs self-disclosure to introduce users to the system's language for producing diagrams. As designers use the direct manipulation tools in CNA, the system discloses related concepts in its language by displaying short Lisp expressions in the bottom pane of Language window. The upper pane of the Language window is where new linguistic commands can be entered into the interpreter. The button labeled with an up-arrow copies disclosures into the interpreter pane so that the revealed expressions can be edited and reinterpreted. Chart 'n' Art also maintains a record of all disclosures in the Transcript window.

In Figure 3 CNA has just disclosed the `make-line` function in response to the use of the mouse-driven line tool.

In CNA nearly every release of the mouse button causes a Lisp expression to appear in the Language window: changing colors, creating, moving, resizing gobjects, etc. Chart 'n' Art prints an extra line break in the Transcript window to visually group disclosed expressions connected with a single operation. If users undo their last direct manipulation action, then press the up-arrow button in the Language window, and then press the return key, the effect is a redo. Designers can then inspect alternate disclosures using the Easier and Harder buttons.

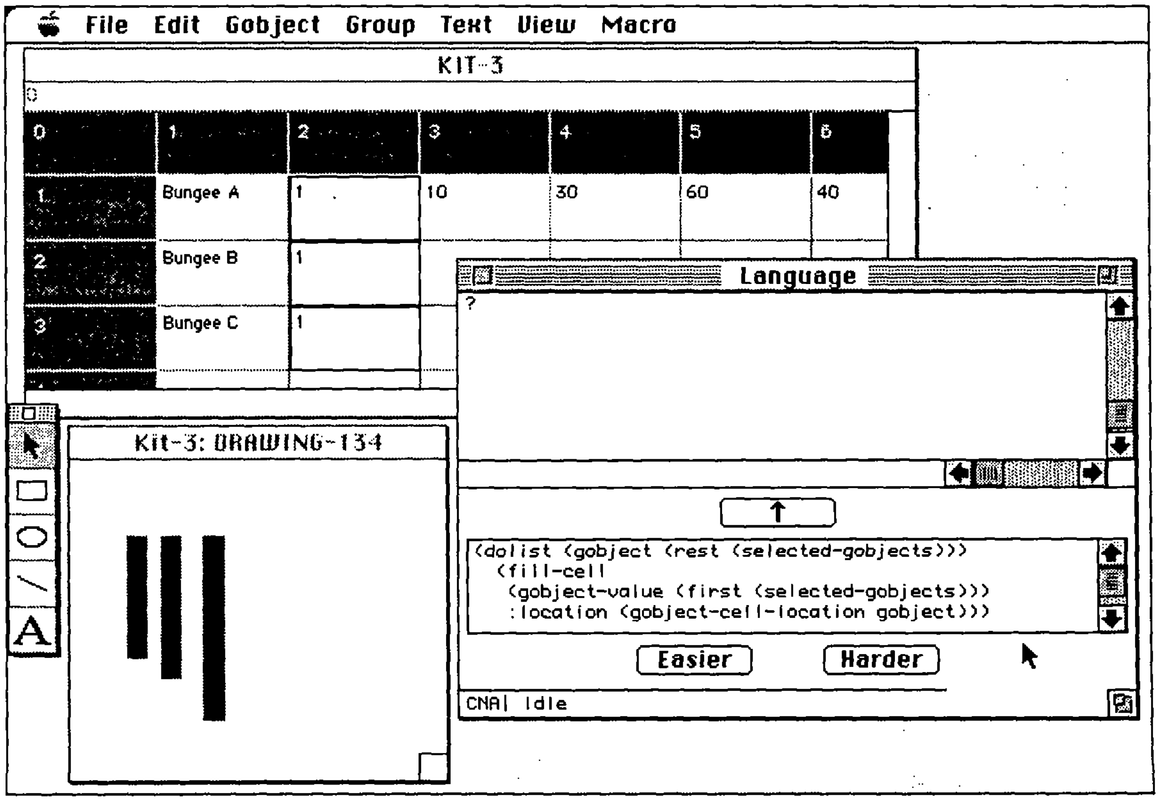
LEARNING FROM CHART 'N' ART DISCLOSURES

To illustrate learning from CNA disclosures, suppose a designer wants to create an inverted column chart where each column grows downward from some fixed height. Such a graph might be appropriate for plotting icicle growth over time or the path of a bungee jumper after leaping from a bridge. How might that designer learn to write a short program in CNA to create this non-standard graph? Perhaps she begins her design task by sketching some exemplars in a drawing window as illustrated in Figure 4. After drawing each rectangle using the mouse, CNA discloses the corresponding `make-rectangle` command. Multiple



```
(make-rectangle :position '(70 259) :size '(12 101))
```

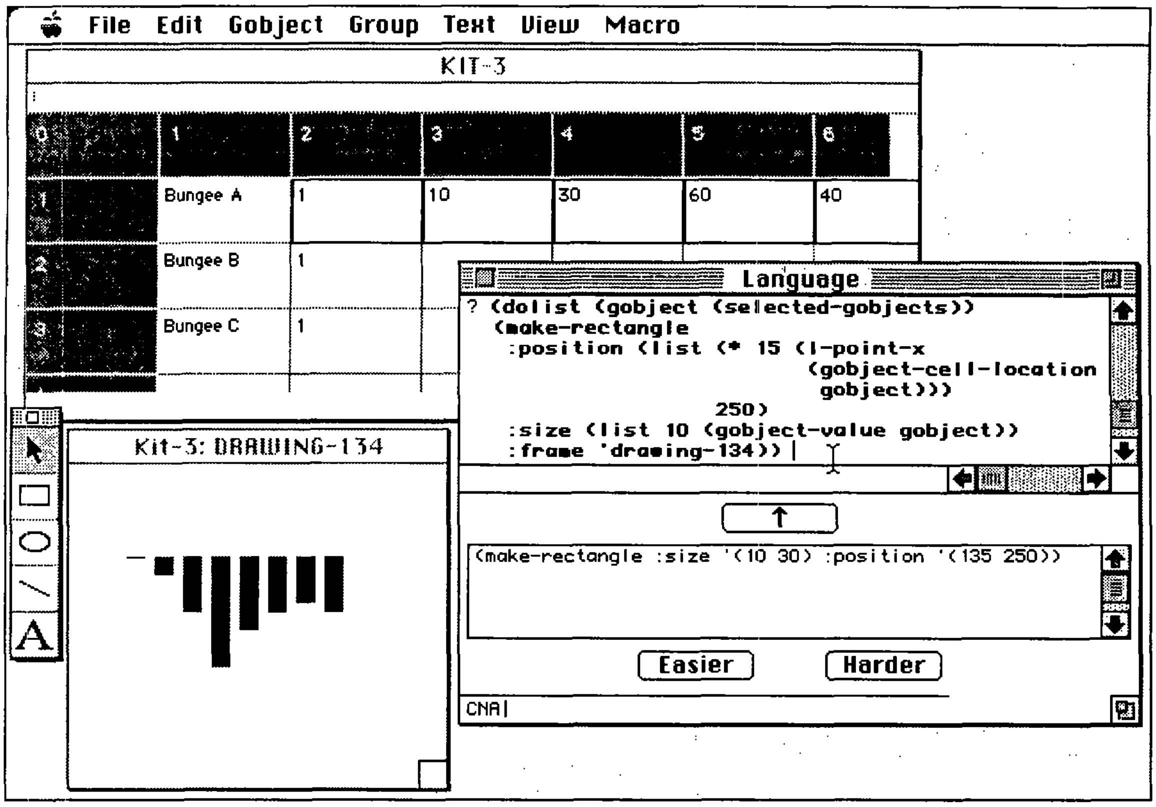
Figure 4. Learning about `make-rectangle` after sketching the initial design for an inverted column chart.



0	1	2	3	4	5	6
1	Bungee A	1	10	30	60	40
2	Bungee B	1				
3	Bungee C	1				

```
(dolist (gobject (rest (selected-gobjects)))
  (fill-cell
   (gobject-value (first (selected-gobjects)))
   :location (gobject-cell-location gobject)))
```

Figure 5. Learning about `dolist` from the Fill Down menu command.



	1	2	3	4	5	6
1	Bungee A	1	10	30	60	40
2	Bungee B	1				
3	Bungee C	1				

```
? (dolist (gobject (selected-gobjects))
  (make-rectangle
   :position (list (* 15 (l-point-x
                           (gobject-cell-location
                            gobject)))
                    250)
   :size (list 10 (gobject-value gobject))
   :frame 'drawing-134)) |
```

```
(make-rectangle :size '(10 30) :position '(135 250))
```

Figure 6. The designer tests her inverted column chart program in the interpreter pane.

make-rectangle disclosures allow her to infer the coordinate space of the drawing window. Pressing the Harder button reveals the optional parameters to the `make-rectangle` function for indicating such properties as `frame` (the gobject's container) and `pen-color`.

While exploring CNA's spreadsheet functionality the designer might encounter further clues into her programming task. Suppose, for example, she enters some comparative bungee jumping data into a kit and uses the Fill Down menu command to replicate the initial value down a column. Selecting Fill Down from the menu causes CNA to disclose the `fill-down` command in the Language window. As shown in Figure 5, pressing the Harder button eventually uncovers the utilitarian `dolist` form underlying the filling operation. This disclosure also suggests how to extract values and coordinates from selected cells using the `gobject-value` and `gobject-cell-location` functions respectively.

The `make-rectangle` and `dolist` disclosures allow the designer to infer the essential building blocks for a short program which generates an inverted column chart from data selected in a spreadsheet. Figure 6 shows just such a program as the designer might enter it in the interpreter pane of the Language window. After selecting the data for Bungee A and then pressing the return key in the Language window, CNA rapidly generates an accurate chart.

SELF-DISCLOSURE GUIDELINES

In developing CNA we have identified six guidelines for the pedagogical use of self-disclosure:

Disclosures should be maximally generalizable

Research on learning from examples suggests users can extrapolate surprisingly detailed and accurate information about a command language from just a few well-constructed exemplars (Anderson, 1987; Lewis, 1988; Lewis, Hair, and Schoenberg, 1989). To maximize the knowledge which can be induced from disclosures, programmable applications should honor certain common user assumptions about the consistency and simplicity of the language interface. Such tools should, for instance, ensure that every component of the disclosed expressions has some obvious connection to components of the action.

As illustrated by the `make-rectangle` disclosure in the Language window in Figure 4, CNA attempts to aid generalizability by using Lisp keyword parameters such as `:size`. The hope is that designers begin to notice that these keywords are descriptors for parameters which invariably follow the keywords. Designers can then better predict the role of each parameter to a function and make generalizations about how it could be used.

The system should facilitate experimentation with disclosures

By allowing designers to easily experiment with disclosures, a programmable application can reduce the effort required to use its language and make it more approachable. Systems can facilitate experimentation by 1) supporting undo in both the direct manipulation and linguistic modes of interaction, 2) enabling disclosed expressions to be easily edited and reinterpreted, so that designers can play with parameter values, and by 3) allowing disclosed expressions to be easily composed into new functions.

Currently, CNA users can experiment with disclosures by selecting lines in the Language window and pressing the up-arrow button. This causes the disclosed text to appear in the interpreter pane at the top of the Language window. In this pane, designers can optionally change parameters and then press the return key to execute the Lisp expression. Users will also soon be able to experiment with functions composed of previously disclosed expressions. The Define Function button in the Transcript window will cause CNA to prompt the user for a name, then generate a function with that name consisting of the currently selected lines in the Transcript.

Self-disclosure should be scaffolded

The language embedded in a programmable application may support programming at more than one level; thus, operations invoked using the mouse might have a number of corresponding linguistic expressions as shown in Figure 7. In this case disclosures should be adapted to the designer's current programming knowledge in order to avoid presenting material that he or she already knows or that is far beyond the designer's present competence. This idea of providing incremental learning support is often called "scaffolding." (Bruner, 1975)

Scaffolding in CNA will be accomplished through a simple mechanism in which the system discloses successively more complex expressions relating to the same direct manipulation command. Initially, disclosures will be presented in their "easiest" form, but CNA will monitor requests by the user for alternate disclosures and adjust the default presentation accordingly. After a designer has been exposed to a particular disclosure numerous times, CNA will attempt to gradually phase in the presentation of a more complex one. Obviously, such a technique oversimplifies the domain of end-user programming knowledge by measuring complexity along a single dimension. Nonetheless, we believe our approach is a practical one and in keeping with the spirit of self-directed learning since the system will simply suggest appropriate disclosures, leaving ultimate control to the designer who can select from alternate representations.

Disclosures should provide coverage of essential programming concepts

A programmable tool can increase designers' awareness of the possible uses for programming by seeding disclosures in such a way that typical sessions with the system will generate information covering fundamental language concepts. For example, if developers know that an average session with their system involves the use of certain menu commands and direct manipulation tools, they should attempt to devise disclosures for each of these mouse actions which cover major language concepts. Following this guideline requires that developers conduct user studies or walkthroughs (Polson, et al., 1992) of their system in order to predict the types of direct manipulations users will employ. Similar studies may also be necessary to determine essential programming expressions in the system's application-oriented language.

```
(choose-color *Purple-Color*)

(choose-color (make-color (color-red 4587685)
                          (color-green 4587685)
                          (color-blue 4587685)))

(choose-color (make-color 17990 0 42405))

(choose-color 4587685)
```

Figure 7. Possible feedback for selecting purple from the color palette.

Chart 'n' Art disclosures currently reveal the gobject maker, mover, and resizer functions, as well as operations for manipulating groups of gobjects and for iterating over kit cells. We plan to use self-disclosure to introduce programming expressions for adjusting the selection and composing functions. Studies are planned to determine the most important programming concepts and to determine patterns of direct manipulation use in CNA.

Designers should be able to specify operations through a combination of direct manipulation and textual commands

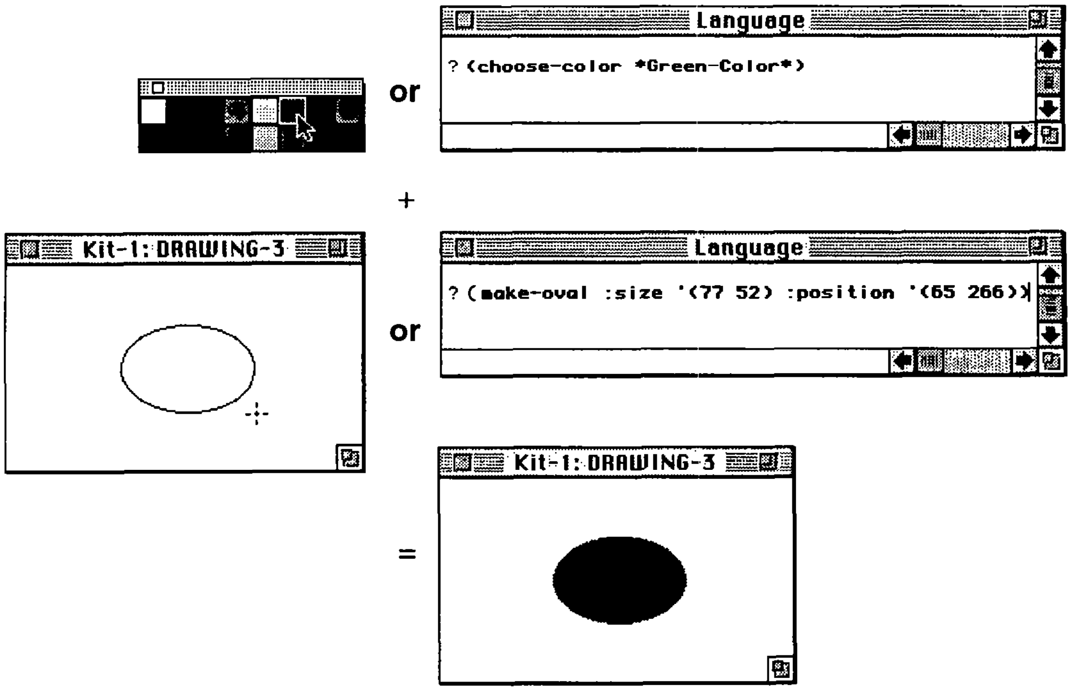
A programmable application should allow designers to use its application's language experimentally, without committing to the exclusive use of the linguistic mode of interaction. For example, designers should be able to select a drawing object with the mouse and then type a command to change the attributes of that object; thus users can play with the language without knowing the programming structures for specifying objects. The ability to interweave interaction modes increases the approachability of the language by allowing users to start employing it with only limited programming knowledge. Interweaving also enables designers to better estimate the effort required for a task involving some programming, since they are likely to already have good estimates for the portions of the task involving direct manipulation.

Chart 'n' Art users can combine direct manipulation with linguistic operations in different ways. For instance, Chart 'n' Art supports the ability to select objects with the mouse and specify the operation on those objects by typing in the upper pane of the Language window. As shown in Figure 8, users can also combine interaction modes when making gobjects. Future versions of the system will allow users to specify coordinate parameters to typed expressions by selecting the screen location with the mouse, as is possible with AutoCAD.

Disclosures should be unobtrusive and browsable

Designers do not always have the time to interrupt their current activity to reflect on disclosures. Even if they do have time to attend to disclosures, designers undoubtedly do not want to have to memorize expressions which might seem useful in the future. In combination, these two factors suggest that disclosures should be both unobtrusive and browsable: unobtrusive, so that designers are not forced to pay attention to the linguistic feedback being generated by the system—a common concern with critiquing systems (Fischer, et al., 1993)—and browsable, so that designers can review previously disclosed expressions and learn at their own pace.

Disclosures in CNA are usually short Lisp expressions displayed in a small font in the Language window and also in the scrollable and resizable Transcript window. If the designer has a large monitor she can easily move this window away from the area of design activity. To facilitate browsing, we are considering making thumbnail-sized before-and-after snapshots which appear in the Transcript window beside each group of disclosures.



```
? (choose-color *Green-Color*)
```

```
? (make-oval :size '(77 52) :position '(65 266))
```

Figure 8. Combinations of direct manipulation and linguistic operations which produce a green oval.

CONCLUSIONS

Developers are beginning to realize that by providing a built-in end-user language they can offer highly functional and flexible tools without having to anticipate the needs of every designer. But programming is no panacea for designers; the reality is that learning an end-user language can be a daunting task. However, the universal human experience of learning to speak suggests that—given the right environment and motivation—humans can master highly complex linguistic operations; and while there are obvious and profound differences between natural and formal language learning, the analogy may nonetheless prove a fruitful one for introducing designers to programming.

The use of self-disclosure, as specified by our guidelines and as instantiated in the language learning opportunities in the CNA prototype system, represent the first step towards a technical and theoretical framework for helping designers become programmers. Preliminary tests with CNA involving two subjects performing a simple drawing task indicate that non-Lisp programmers can infer from disclosures a subset of the vocabulary of CNA's domain-enriched language as well the basic open-parenthesis-operator-operand-close-parenthesis syntax. Clearly, more thorough assessment and refinement of self-disclosure mechanisms will be necessary to validate our approach.

In this context, it is worth addressing the "language question"—i.e., our potentially controversial choice of Lisp as the end-user programming language for CNA. There are several points to make here. First, we wish to stress that CNA is designed to support programming "in the small"—a style of programming in which short, "domain-enriched" expressions accomplish powerful ends within an application. The same sort of philosophy might arguably be attributed to languages such as Mathematica[8] (in which short procedure calls accomplish sophisticated mathematical or graphical tasks).

Viewing end-user programming in this light, there need not be a sharp division between "professional" languages and "end-user" languages: in many cases, the former may be tailored for use in a particular application or domain (as suggested by examples such as Visual BASIC and AutoCAD's dialect of Lisp). This is not to say that questions of language learnability should be ignored—but our belief is that the notion of self-disclosure can lead language designers to reexamine these questions, focusing on those language features that lend themselves best to incidental learning. Finally, the techniques of self-disclosure described in this paper, while illustrated by a particular Lisp-based system, are also intended for broad applicability: such tenets as unobtrusiveness, browsability, coverage, and so forth are likely to prove useful in applications based on a wide variety of end-user programming languages.

Our belief is that a tool such as CNA offers the potential for supporting the type of professional education that Schön (Schön, 1983) writes about—an education of "reflection-in-action," of interweaving well-practiced (or near-automatic) activity with pauses for conscious reflection upon one's work. This is a consequence of the tool's situated presentation of language concepts: the designer does not have to structure her activity into artificially distinct periods of language-learning followed by (almost necessarily impoverished) language use. Rather, the designer develops expertise in the context of meaningful projects. Moreover, CNA aims for a more ambitious type of learning than that generally

[8]Mathematica is a registered trademark of Wolfram Research, Inc.

associated with context-sensitive help systems: the system is not limited to informing the user of an important vocabulary item or alerting her to an immediate syntactic problem. Instead, the type of learning supported by CNA is a larger-scale (and longer-term) learning of a new medium. The distinction that we are after here is suggested by Norman's (Norman, 1993, p. 28) taxonomy of learning, in which restructuring is distinguished from accretion: whereas the latter involves learning new particular facts or vocabulary items within some well-understood general framework (the type of learning associated with context-sensitive help), the former involves the development of that general framework itself. In the case of CNA, our intent is to provide designers who have never programmed with a setting in which to adapt to an entirely new (and, as noted, often intimidating) medium.

Finally—stepping back from the particular system discussed here—self-disclosure may have applications for users other than designers, to systems other than programmable applications, and to domains other than programming language learning. It is possible, for example, that computer science students could benefit from the use of self-disclosure when learning new programming languages, perhaps embedded in interpreters for languages they already know. Another possibility is that beginning designers could learn design heuristics from a self-disclosing information graphics program that revealed presentation techniques while they chose from high-level charting operations. Self-disclosure may prove to be a powerful and general strategy for presenting new knowledge to users performing any number of computer-based tasks.

ACKNOWLEDGMENTS

We would like to thank Tom Landauer, Clayton Lewis, Jim Martin, and Nancy Songer for their comments on a draft of this paper. This research was supported by the National Science Foundation under grant number RED-9253425.

REFERENCES

Anderson, J.R., and B.J. Reiser (1985). The LISP Tutor. *BYTE* 10 159-175.

Anderson, J.R. (1987). Causal analysis and inductive learning. *Proceedings of the Fourth International Machine Learning Conference*, 288-299.

Bruner, J. S. (1975). The ontogenesis of language. *Journal of Child Language* 2 1-19.

Carroll, J.M., and M.B. Rosson (1987). Paradox of the Active User. In *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. Edited by J. M. Carroll. 80-111. Cambridge, MA: The MIT Press.

DeJong, G. (1983). An approach to learning from observation. *Proceedings of the International Machine Learning Workshop.*

DiGiano, Chris, and Mike Eisenberg (1995). *Supporting the end-user programmer as a lifelong learner*. Department of Computer Science, University of Colorado at Boulder. Report #CU-CS-761-95.

Eisenberg, Mike (1995). Programmable Applications: Interpreter Meets Interface. *SIGCHI Bulletin* 27 (2): 68-93.

Fischer, G. (1987). A Critic for LISP. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy)*. Edited by J. McDermott. 177-184. Los Altos, CA: Morgan Kaufmann Publishers.

Fischer, Gerhard, Kumiyo Nakakoji, Jonathan Ostwald, Gerry Stahl, and Tamara Sumner (1993). Embedding Computer-Based Critics in the Contexts of Design. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*. 157-164.

Gantt, Michelle, and Bonnie A. Nardi (1992). Gardeners and Gurus: Patterns of Cooperation among CAD Users. In *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*. 107-117.

Gleitman, Lila R. (1994). The structural sources of verb meanings. In *Language Acquisition: Core Readings*. Edited by P. Bloom. 174-221. Cambridge, MA: The MIT Press.

Lewis, C. (1988). Why and How to Learn Why: Analysis-Based Generalization of Procedures. *Cognitive Science* 12 211-256.

Lewis, Clayton, D. Charles Hair, and Victor Schoenberg (1989). Generalization, Consistency, and Control. In *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems*. 1-5.

Nardi, B.A. (1993). *A Small Matter of Programming*. Cambridge, MA: The MIT Press.

Nardi, Bonnie A., and James R. Miller (1991). Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *International Journal of Man-Machine Studies* 34 (2): 161-184.

Nishioka, Fumihiko (1992). *Diagram Graphics*. Tokyo: P•I•E Books.

Norman, Donald A. (1993). *Things that make us smart*. Reading, MA: Addison-Wesley Publishing Company, Inc.

Polson, Peter G., Clayton Lewis, John Rieman, and Cathleen Wharton (1992). Cognitive Walkthroughs: A Method for Theory-Based Evaluation of User Interfaces. *International Journal of Man-Machine Studies* 36 (5): 741-773.

Richards, I. A. (1973). *English through pictures*. New York: Pocket Books.

Schön, D.A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books.

Stallman, R.M. (1981). EMACS, the Extensible, Customizable, Self-Documenting Display Editor. *ACM SIGOA Newsletter* 2 (1): 147-156.