Introducing designers to programming through self-disclosing tools

Chris DiGiano

Technical Report #CU-CS-770-95

Introducing designers to programming through self-disclosing tools

Chris DiGiano

Overview

Programmable tools for design offer users an expressive new medium for their work, but current support for becoming acquainted with their tool's language is limited. The goal of my proposed dissertation research is to devise mechanisms that can be embedded in a programmable tool to support the acquisition of its language. In this proposal *I hypothesize that a programmable design tool which discloses programming language information relevant to users' current activity can be an effective means of introducing them to the language*. I identify the essential characteristics of this *self-disclosure* technique and the type of learning it supports. This leads to guidelines for the use of self-disclosure specifically for embedding language learning opportunities into design tools. I present a prototype information graphics design tool called *Chart 'n' Art* for exploring self-disclosure techniques. Finally, I propose a method for comparing the insights gained through self-disclosure to other means of instruction.

Table of contents

1.0 Introduction

In the 1950's, poet, educator and scholar I.A. Richards published a series of books called *Language through Pictures* (Richards, 1973) as a teaching tool for second language learners. Each page 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 situations to more complex ones, 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 interesting question for the area of computer science education: Can similar approaches be found to support the acquisition of programming languages? This proposal 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.

For my dissertation I will focus on supporting designers—architects, graphic artists, and dress makers to name a few—learning to write short programs for their creative activity. Increasingly, designers are turning to computers to aid their tasks. For some forms of design, such as architecture, computers have become indispensable tools. Nonetheless, the utility of a computer-based design tool is limited by the coverage it provides over the functionality required by the user. One response to this limitation is *programmable applications* (Eisenberg, 1991). Programming offers the opportunity to transcend the built-in functionality of software, empowering users to be more creative and expressive.

For most designers programming is not a natural activity. Rarely is programming part of their formal training. And yet many designers have an interest in increasing their skills and have some sense that programming could be beneficial. Others might be interested in learning programming if only they knew its potential benefits for their design activity. The challenge then is informing designers of the utility of programming and supporting them in their pursuit of programming expertise. These challenges stand in the way of a user eventually accepting an application-oriented language and employing it creatively, and are in fact the central problem for the entire field of programmable applications (DiGiano and Eisenberg, submitted 6 October 1994).

2.0 Introducing designers to programming

In this section I explore the issues involved in introducing programming to designers. First, I describe who designers are and what it is they do, and then discuss how programming might be useful to them, and what form a programmable tool might take. Finally, I conclude by identifying barriers to designers using programming in their creative process.

2.1 Who are designers?

A designer is one who devises or creates the form or structure of an artifact, whether it be a work of art such as a dress or a scheme such as for stage lighting. According to Simon, designers devise "courses of action aimed at changing existing situations into preferred ones." (Simon, 1981, p. 129) A key feature of their activity is that it rarely follows a fixed procedure, but rather involves searching a space of alternatives for a candidate which satisfies design criteria. These criteria often reflect designers' tacit informal knowledge of their domain acquired by *reflecting-in-action* (Schoen, 1983, p. 54). Designers' training is often highly specialized, involving several years learning the notation and tools of their domain. Apprenticeships under master designers, especially in the areas of graphic and architectural design are common forms of training.[1]

Although my proposed dissertation research is relevant to a variety of design domains, I have chosen to focus on information graphics. Information graphics designers create visual presentations of data whether they be financial, meteorological, chronological, or astronomical. Criteria used by designers to evaluate information graphics include expressiveness—whether designs portray the desired information, and effectiveness—whether designs exploit the capabilities of the output medium and the human visual and conceptual systems (Mackinlay, 1986). Tufte in his book *Envisioning Information* describes the essence of information graphics:

> We envision information in order to reason about, communicate, document, and preserve that knowledge—activities nearly always carried out on two-dimensional paper and computer screen. Escaping this flatland and enriching the density of data displays are the essential tasks of information design... all the history of information displays and statistical graphics—indeed any com-

1. Simon points out in *The Sciences of the Artificial*, (Simon, 1981, p.129) that design education for many domains such as medicine and engineering has shifted toward decontextualized instruction. Nonetheless, the "softer" design professions such as graphic design and architecture have resisted this shift.

munication device—is entirely a progress of methods for enhancing density, complexity, dimensionality, and even sometimes beauty (Tufte, 1990, p. 33).

2.2 A case for programming in design

If ease of use was the only valid criterion, people would stick to tricycles and never try bicycles. —Doug Engelbart

A programming language embedded in a design tool gives designers access to potentially large libraries of commands, and empowers them with the ability to compose their own functions which encapsulate complex or iterative behavior. Thus programming offers designers the opportunity to transcend the built-in functionality of software, enabling entirely new kinds or processes of designs that were not otherwise possible or practical. The following scenario illustrates the need for programming in the design of non-standard information graphics.

2.2.1 Alice and the Denver-Lyons bus route[1]

Alice is a free-lance graphic artist recently hired by a transit company to design a schedule for a new bus route whose timetable is shown in Table 1. Her task is to produce an informative design which indicates arrival/departure times, relative speeds of different buses, where transfers can be made, and how long buses wait at particular stops, among other things. Alice decides her design should consist of two parts: something similar to Table 1 to help answer specific arrival/departure questions, and a time-space diagram similar to that in Figure 1 to help quickly answer speed, transfer, and stop questions. Notice the following features of her time-space diagram:

- Both directions of the route can be represented in a single diagram.
- Relative distance between stations is captured in the distance between stop labels.
- Speed of the buses can be determined by the slope of their lines. The steeper the slope, the faster the bus.
- A wait at a stop causes a bus's line to appear horizontal. The length of that line is proportional to the amount of time waiting at the stop.

1. This scenario is based on a real design problem illustrated in *Envisioning Information* for depicting train activity over the Hava railroad line, Soerabaja-Djokjakarta. The incredibly complex one-page solution drawn in November 1937 cleverly depicts 16 variables concerning the movement of some 40 trains over a 24 hour period.

- Possible transfers from a bus arriving at a particular stop can be found by scanning across the stop's row to the right of the line representing arriving bus.
- Passing buses are indicated by intersecting lines. (this information might be useful to bus operators)

TABLE 1. The original bus schedule data given to Alice

Lyons to Denver South Monday-Friday

Lyons	Boulder		Westminster		Denver
depart	arrive	depart	arrive	depart	arrive
800AM	840	900	930	931	959
905	935	936	1010	1011	1030
1000	1040	1100	1130	1131	1159
1105	1135	1136	1210PM	1211	1230
1200	1240PM	100	130	131	159
105PM	135	136	210	211	230
200	240	300	330	331	359
305	335	336	310	311	330
400	440	500	530	531	559
505	535	536	610	611	630

Denver to Lyons North Monday-Friday

Denver	Westminster		Boulder		Lyons
depart	arrive	depart	arrive	depart	arrive
808	828	835	909	915	1002
900	928	935	1005	1015	1100
1008	1028	1035	1109	1115	1202PM
1100	1128	1135	1205PM	1215	100
1208PM	1228	1235	109	115	202
100	128	135	205	215	300
208	228	235	309	315	402
300	328	335	405	415	500
408	428	435	509	515	602
500	528	535	605	615	700

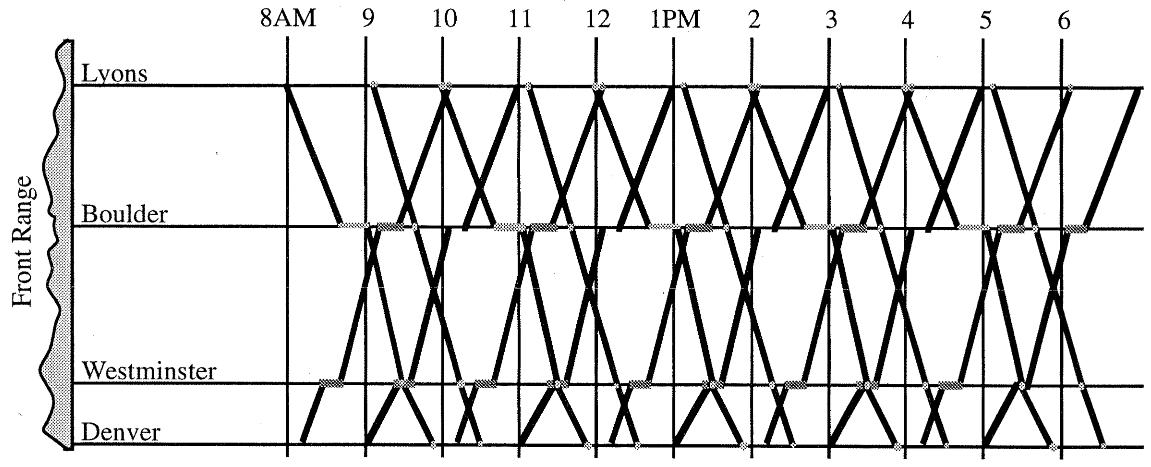
2.2.2 Limitations of traditional tools

The major drawback to the otherwise elegant and information rich presentation in Figure 1 is the difficulty in producing it. No charting software that is currently available has a "transportation schedule" chart type or any other type for that matter which could reproduce the details of Alice's concept. DeltaGraph Pro's[1] X-Y Line chart would generate the

1. DeltaGraph is a registered trademark of DeltaPoint Corporation.

connected lines but only after one mapped times to X values and the stop names to Y values. Even then, the graph would be missing the shading used to highlight each stop. On the other hand, Alice could use a drawing package such as MacDraw[1] to reproduce the subtle nuances of her diagram, including the stop shading and the image of the front range on the left side. However, the drawing program approach requires that Alice tediously maps the times for each bus onto a location in the time-space graph, possibly leading to inaccuracies.

FIGURE 1. Alice's information-rich design concept



2.2.3 Programmable tools for design

A solution to Alice's diagram-production dilemma is an end-user programmable design tool. Programmable systems allow *end-users*—people with specific tasks in mind, and little or no intrinsic interest in computers—to employ a formal language to specify information structures and transformations on those structures. Such programmable tools are becoming more and more prevalent. Microsoft, for instance, has begun integrating its Visual BASIC language into most of its personal productivity software including Word and Excel.[2]

For my dissertation I will focus on a specific type of end-user programmable system, the *programmable application,* which combines a direct manipulation interface with a domain-specific programming language

1. MacDraw is a registered trademark of Claris Corporation.
2. Microsoft, Visual BASIC, Word, and Excel are registered trademarks of Microsoft Corporation.

interpreter. Programmable applications include the drawing program SchemePaint (Eisenberg, 1991), the computer aided design (CAD) package AutoCAD,[1] the Microsoft Excel spreadsheet, and the symbolic manipulation tool Mathematica.[2] A programmable tool for our end-user Alice would ideally have direct manipulation operations for entering tables of numbers and for producing freehand artwork such as the Front Range. Likewise, the tool's application-oriented language would include functions for perusing the data in a table and mapping them to locations in a drawing and to various choices of shading and coloring.

Alice could use her programmable application's language to compose a small procedure which looked up the departure time for a bus leaving Lyons and started a polyline at the corresponding point along the latitude line for Lyons. Another procedure could add a point to the polyline corresponding to the bus's next stop. By calling this procedure repeatedly, Alice could generate a complete time-space representation for the bus's route and, by refining the procedure, she even could produce shaded lines during stops. Alice could either generalize or copy and modify these procedures to handle buses going in the opposite direction.

2.3 Challenges to the use of programming in design

The above scenario highlights the important role programming can play in design activity. However, there are critical barriers which designers must overcome in order to realize the power of programming. Some of these barriers are imposed by the programmable systems themselves, while others are the result of individual factors. Experience with end-user modifiable environments such as the Athena X Window System indicates that users will not take advantage of programmability unless these barriers are somehow addressed (Mackay, 1991). Below I outline the major issues concerning the usefulness and usability of programmable tools.

2.3.1 Programming language issues

Approachability. The initial experience with a language has a large influence on whether designers will continue to use it. 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*.[original italics]" (Nardi, 1993, p. 45) Approachability, however, must be balanced with support for designers who gradually become more sophisticated programmers.

1. AutoCAD is a registered trademark of Autodesk Corporation.
2. Mathematica is a registered trademark of Wolfram Research Corporation.

Domain orientation. The success of a programming language for a designer is largely dependent on how well it is adapted to his or her design domain. One of the most troublesome tasks for programmers, especially beginners, is mapping problems to the solution space offered by a programming language (Eisenberg and Fischer, 1994; Fischer, 1987).

Language granularity. High-level programming expressions are only useful when they exactly match the designer's tasks, but low-level expressions tend to be more abstract and introduce the additional problem of composition. For beginning programmers, "it is hard to see what combination of primitives will produce the correct task-related behavior" (Lewis and Olson, 1987)

2.3.2 Individual user issues

Time for learning. Perhaps the most critical issue is whether designers have time to begin acquiring a programming language. Designers face a "production paradox" (Carroll and Rosson, 1986): learning the language requires time but using the language can potentially save time. Mackay concludes her study by suggesting that users must choose between "activities that accomplish work directly and activities that may increase future satisfaction or productivity." (Mackay, 1991)

Knowing the possibilities. Awareness of the functionality offered by programming is critical before most designers will consider using it in their creative activity. Knowledge of a system's capabilities is in fact a common issue for any complex system (Fischer, 1991a). This problem of "cognitive bootstrapping" (Resnick, 1989) is a challenge for tool developers since they cannot assume awareness will naturally develop as a result of designers' curiosity about their system.

Estimating the effort. Programming is potentially a significant digression from a designer's task at hand, especially if it involves having to learn new programming knowledge. Mackay's study of Athena users suggests a large number of them are "not willing to risk spending an unknown amount of time on customization."(Mackay, 1991) An important issue, then, is whether designers can estimate the effort involved in using programming in a task.

Mapping instruction to authentic use. Instructional resources such as manuals and training courses are often available for programmable tools. However, the effectiveness of these resources is limited by how easily designers can map instructed knowledge to knowledge-in-use (Resnick, 1989). This problem is also confounded by the domain-orientation issue, since no matter how appropriate the instruction, designers will have difficulty applying a poorly-oriented language to their authentic tasks.

2.4 The problem: lack of support for beginning end-user programmers

Although end-user programmable systems represent a burgeoning class of software, support for 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 of success within a few hours.

Printed tutorials, on-line tutoring programs, and training classes are some of the few support mechanism widely available to users learning programmable tools.[1] 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 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 MacroMedia Director for its Lingo language[2] 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.

1. Experimental intelligent tutoring systems such as the Lisp Tutor (Anderson and Reiser, 1985) could hardly be called "widely available," but they do nonetheless suffer from some of the same problems as traditional on-line tutorials.
2. Lingo is a registered trademark of MacroMedia Corporation.

3.0 Using self-disclosure to gently introduce designers to programming

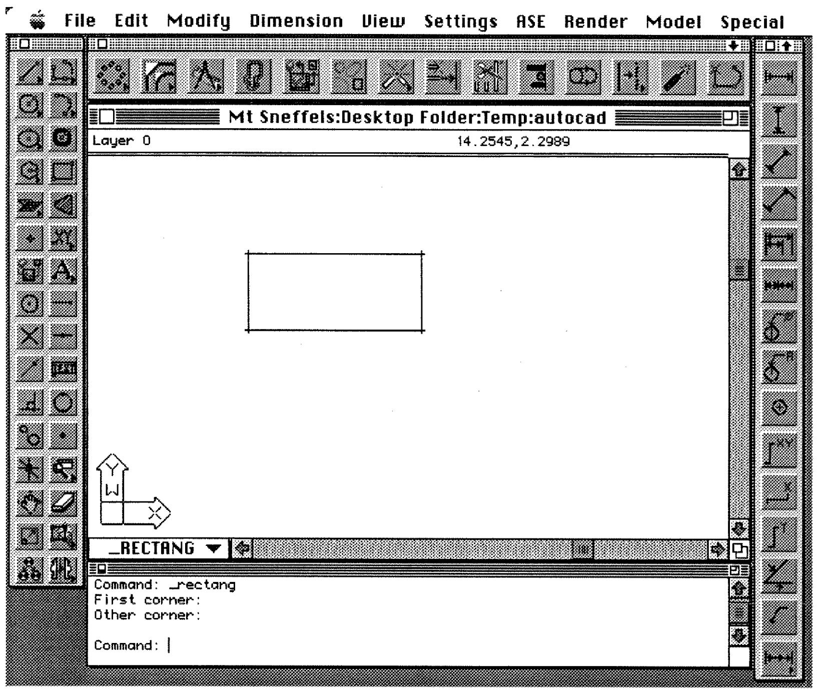
Experience with existing resources for users learning their tool's application-oriented language suggests effective support mechanism must 1) reduce both time and effort required by the user, 2) facilitate estimation of costs and benefits, 3) minimize prerequisite programming knowledge, and 4) situate learning in authentic use. In this section I describe an approach which, like 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"[1] to gently introduce designers to programming. That is, the programmable tool "discloses" elements of its language to designers as they use it. First, I present an anecdote which illustrates the effective use of self-disclosure, and then I characterize both the type of learning which took place and the essential system behavior which enabled it. Finally, I present guidelines for tuning self-disclosure to best fit the needs of designers learning programming for the first time.

3.1 Self-disclosure in action

One of the earliest examples of programmable design tools is the still-popular computer aided design system, AutoCAD, shown in Figure 2, 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. AutoCAD cleverly coordinates these two modes of interaction. For example, mouse clicks on one of its extensive toolbars causes the system to disclose the equivalent historical command name in the Command window. Users can also enter coordinate parameters either by typing their values or by using the mouse to select screen locations.

1. Computer scientists have used the term "self-disclosure" to describe interactive techniques for acquainting users to stylus gestures recognized by pen-based systems (Venolia, 1994). Psychologists use "self-disclosure" to describe the act of publicly expressing personal thoughts or feelings. (Derlega and Berg, 1987)

FIGURE 2. AutoCAD's self-disclosing interface. Drawing a rectangle with the mouse reveals the command `_rectangle` in the bottom Command window.



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

Command: |
```

I once interviewed an experienced architect and graphic designer who had started using AutoCAD several months before 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 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 system's command language. But 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.

3.2 Learning from disclosures

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 make useful generalizations (DeJong, 1983; Gleitman, 1994). The AutoCAD designer, for example, 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.

Learning by observation is not a new idea. In 1690 British philosopher John Locke wrote about how observation plays a key role in human language acquisition:

> If we will observe how children learn languages, we will find that to make them understand what the names of simple ideas or substances stand for, people ordinarily show them the thing whereof they would have them have the idea; and then repeat to them the name that stands for it, as “white,” “sweet,” “milk,” “sugar,” “cat,” “dog.” (Locke, 1964)

Although linguists have identified other mechanisms employed by language learners most agree that our first steps towards acquisition involve forming word-to-world mappings through observation. Learning by observation can be considered a specialized form of learning by example (Hayes-Roth, 1978) involving the analysis of pairs of short-term perceptible events. Types of paired events include an action by the learner and the environment’s reaction, two closely-timed actions by another agent (as described by Locke), or an environmental event followed by a reaction by another agent.

Learning by observation seems to occur most readily when new knowledge is appropriate to the learner’s current level of competence. For example, it is widely believed by linguists that graduated child-directed speech or “motherese,” in which adult speech is adapted to an infant’s abilities, helps accelerate mother tongue acquisition (Gleitman, 1984). Nonetheless, learning by observation can be a powerful strategy even without well-structured input. Consider how infants manage to develop word meanings despite the fact that their language input is not particularly organized to facilitate the categorization of objects. For instance, the language learner will eventually acquire the appropriate concept of “dog” most likely without anyone having pointed to a series of different kinds of dogs saying the word “dog” after each example. Instead, learners seem to periodically revise stored word meanings as the terms are presented in an essentially random order. Certainly in the AutoCAD anecdote, the system made no attempt to structure its command language feedback, and yet the designer made the correct assumptions.

3.3 Characteristics of self-disclosing systems

AutoCAD's disclosures allowed the designer I interviewed to begin learning the command language by observation. Users of many spreadsheet programs 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 cell immediately below the column, thus revealing part of the formula language to the user. However, it is doubtful that developers of these tools specifically provided 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.

What are the attributes of a programmable application which enable language learning to occur even when pedagogy is not the main intent of the system's use of self-disclosure? I characterize such systems as having three essential properties:

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 to the user 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.

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, 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.

3.4 Guidelines for the effective pedagogical use of self-disclosure

Ensuring that a system exhibits the three self-disclosure properties is only the first step in addressing the needs of designers learning their tool's application oriented language. For example, although AutoCAD fits each property, the designer in the anecdote still had to digress from her task to read manuals on composing operations using the command language. I claim that the issues from Section 2.3 can be more thoroughly addressed by shrewdly designing disclosures and the language environment for using them. The primary goal of my proposed dissertation research is to develop guidelines for the pedagogical use of self-disclosure in programmable design tools. *My hypothesis is that by using these guidelines self-disclosure can be an effective instructional technique which addresses the specific needs of designers learning to program their tools while performing authentic tasks.* Below is a preliminary set of guidelines, some of which are already followed in existing programmable applications.

3.4.1 Disclosures should be maximally generalizable

Disclosures offer the designer the opportunity to begin constructing bit by bit a vocabulary of application-oriented programming terms and a model for the general schema of the built-in language. Research on learning from examples suggests users can extrapolate surprisingly detailed and accurate information about a command language from just a few exemplars (Anderson, 1987; Lewis, 1988; Lewis, Hair, and Schoenberg, 1989). But these exemplars should be designed to facilitate the most useful generalizations.

Promising strategies for structuring generalizable disclosures are found in the causal analysis and program comprehension literature. Lewis, Hair, and Schoenberg posit that users employ causal analysis to make generalizations about a new interface. Such analysis is facilitated by systems which honor certain common user assumptions about the consistency and simplicity of the interface. Their work suggests that a self-disclosing design tool should, for instance, ensure that every component of the expressions has some obvious connection to components of the action. Research in program comprehension indicates that "beacons" or "signals" such as appropriately-named functions and the consistent use of capitalization can contribute to the readability and understandability of code (Gellenbeck and Cook, 1991a; Gellenbeck and Cook, 1991b).

3.4.2 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 functions.

AutoCAD provides none of the above support for the beginning end-user programmer, which may partly explain why the designer from the anecdote resorted to language reference manuals so soon.

3.4.3 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. In this case disclosures should be adapted to designer's current programming knowledge in order to avoid presenting material which the designer already knows or which is far beyond the designer's present competence. This idea of providing incremental learning support is often called "scaffolding." (Bruner, 1975) For example, if Alice from Section 1 uses a menu command to align the left sides of each bus stop name in Figure 1 the system might respond with any one of the possible disclosures in Table 2 depending on the concepts it determined were appropriate for her

Programmable applications can employ a user model (Wenger, 1987, p. 126-135) to represent competence levels for each designer, and as shown in Figure 2 to filter disclosures. The figures illustrates how a user action is handled by the tool's interface which provides a collection of feedback possibilities such a, b, and c in Table 2 to an "expander" module. Each possibility is annotated with some indication of the programming competence required to understand it. The expander attempts to derive additional alternate possibilities automatically. For instance, option c in Table 2 could have been derived from option b, simply by substituting list accessors for coordinate accessors. Finally, an arbiter module employs the user model to choose the most appropriate of all the disclosure possibilities..

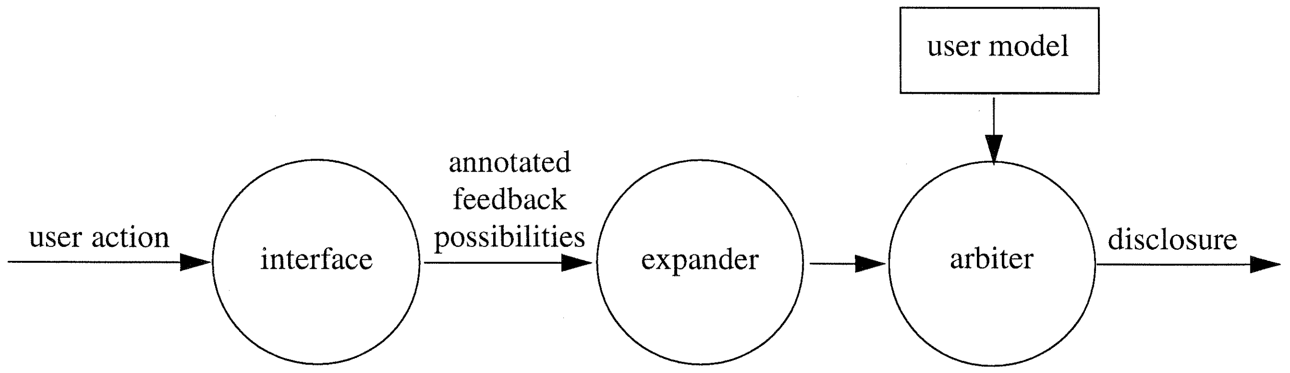
TABLE 2. Possible feedback for the align left menu command.

Option	Possible disclosure	Relevant concepts
a	`(align-gobjects :how :left)`	keyword parameters

b	`(dolist (Gobject (selected-gobjects))`
	`  (if (> (lpoint-x (gobject-position Gobject)) 58)`
	`    (move-gobject Gobject`
	`                 (make-lpoint`
	`                   220`
	`                   (lpoint-x`
	`                 (gobject-position gobject))))))`
		coordinate constructors
		coordinate accessors
		iteration over selections
		conditionals

c	`(dolist (Gobject (selected-gobjects))`
	`  (if (> (first (gobject-position Gobject)) 58)`
	`    (move-gobject Gobject`
	`                 (list`
	`                   220`
	`                   (first`
	`                 (gobject-position gobject))))))`
		iteration over selections
		conditionals
		lists as coordinates

FIGURE 3. A data flow model of graduated self-disclosure.



3.4.4 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 users studies or walkthroughs 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.

3.4.5 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 "intertwine" interaction modes increases the approachability of the language by allowing users to start employing it with only limited programming knowledge. Intertwining 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.

3.4.6 Disclosures should be subtle 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 subtle and browsable. Subtle, so that designers are not forced to pay attention to the linguistic feedback being generated by the system. Browsable, so that designers can review previously disclosed expressions and learn at their own pace. An example mechanism to support browsable disclosures is the scrollable Command window in AutoCAD as shown in Figure 2. Unfortunately, AutoCAD disclosures do not persist between sessions.

3.5 Related learning theories

The above self-disclosure guidelines represent a preliminary framework for make learning programming a natural part of design activity. The self-disclosure approach is related to several other educational modalities which I present below.

Learning by observation. Self-disclosure allows designers to learn by observing patterns in programming expressions generated in response to their own interactions with the tool. As mentioned earlier, humans seem uniquely adapted for this kind of learning as evidenced by our impressive ability to acquire languages. One might even consider disclosure as being analogous to caregivers' spoken feedback to an infant exploring the world.

Increasingly complex microworlds and ZPD. Scaffolded self-disclosure seeks to adapt feedback appropriately for the present competence of the designers. This technique resonates with approaches to incremental instruction such increasingly complex microworlds (Burton, Brown, and Fischer, 1984) and methods based on Vygotsky's idea of zones of proximal development (ZPD) (Vygotsky, 1978). A ZPD delineates an appropriate level of instruction and a degree of reliance on a supporting context. Educators have applied ZPD with some success in the classroom (Brown, et al., 1993) and have also integrated ZPD into computer-mediated educational environments such as CSILE (Scardamalia and Bereiter, 1991).

Situated learning. Disclosed programming information is naturally situated to the designer's current task, since the feedback is in direct response to user activity. Situated learning represents the thrust of promising instructional theories such as cognitive apprenticeship (Collins, Brown, and Newman, 1989), discovery learning (Elsom-Cook, 1990), and learning on demand (Fischer, 1991b). These methods seek to provide learning opportunities directly related to what the learner is doing.

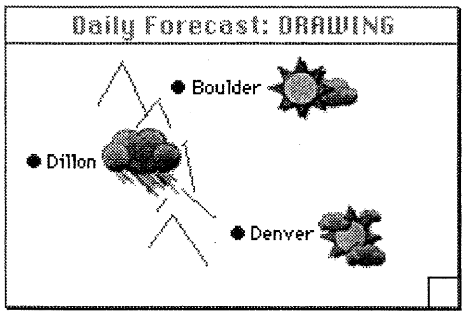
Optimal flow. Disclosures are ideally short, subtle, and unobtrusive. Designers can choose to ignore their tool's feedback or do away with it altogether. Thus self-disclosing tools allow designers to concentrate almost continuously on their work, a requirement for what Mihaly Csikszentmihalyi calls optimal flow (Csikszentmihalyi, 1990). Norman applies this idea to computer based tools:

> All attention should be concentrated upon the task itself, not upon the tool. When the tool calls attention to itself, that creates a breakdown in the work flow. Tools should stay in the background, becoming a natural part of the task... The tools, the person, and the task meld into a seamless whole (Norman, 1993).

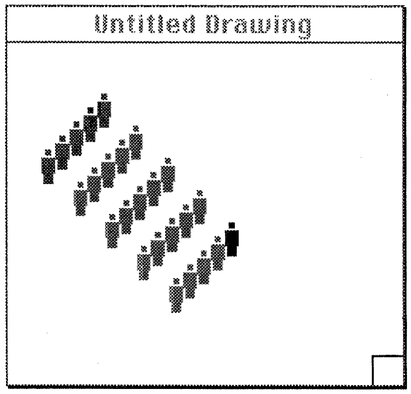
4.0 Chart 'n' Art: a prototype self-disclosing design tool

Chart 'n' Art (CNA) is a programmable information graphics tool I have begun developing in order to test and refine guidelines for the use of self-disclosure. One of CNA's goals is to provide sufficient functionality to support information graphics designers in creating non-standard diagrams such as those depicted in Figure 4 which could not easily be made with commercial packages such as DeltaGraph Pro. Figure 4a 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 4b is a reconstruction of a unique time line featured in *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. Red figures show generations using written language—they number some 200, but in Figure 4b all but 5 have been clipped. Orange figures represent the use of the printed books—some 19 generations. The single blue figure symbolizes the current use of computers. Again, a short function was written, this time to translate essentially a diagram "key" into a time line.

FIGURE 4. Some non-standard diagrams produced by Chart 'n' Art



a



b

4.1 The Chart 'n' Art interface

Like many charting programing 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.

4.1.1 Chart 'n' Art direct manipulation operations

Designers can employ direct manipulation to perform standard drawing and spreadsheet operations. Using a tool palette and color palette, designers can created 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.

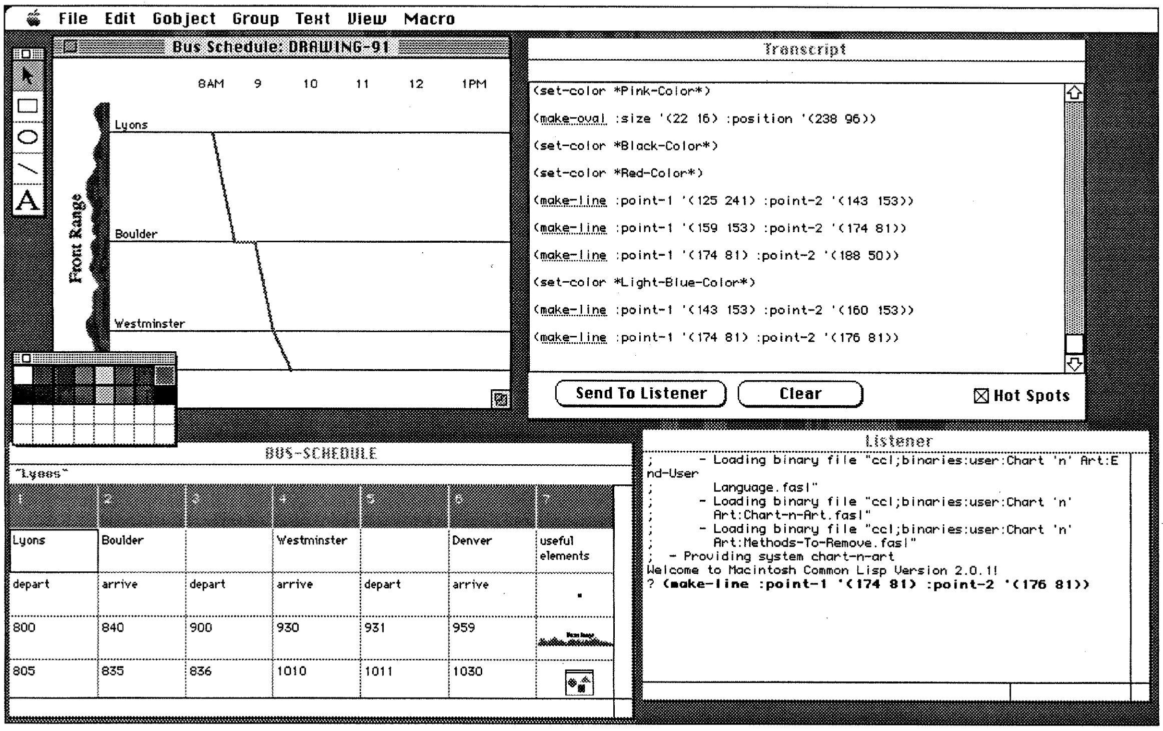
4.1.2 The Chart 'n' Art application-oriented language

The CNA application-oriented language is an extension of Lisp designed to support the kind of data-to-picture translations shown in Figure 4. The language can be used to create or modify spreadsheet cells, introduce new 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-rect-angle` 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` and `select-down` for affecting a selection relatively, and `select-at` for affecting it absolutely.

4.1.3 Chart 'n' Art windows

To illustrate CNA's self-disclosing features I return to the bus schedule scenario in Section 2. Figure 5 depicts CNA's main windows and palettes as they might appear during one of Alice's first sessions. Alice has entered the bus departure and arrival times into a kit called Bus Schedule. The kit also includes a graphical element, a picture of Colorado's Front Range. A drawing window labeled "Bus Schedule: DRAWING-91" shows the beginnings of the line drawing for depicting a bus's path through space and time. By this window are the standard drawing program palettes for selecting drawing tools and colors. The window labeled "Transcript" is where disclosures appear—note, CNA has already revealed the `make-line` function in response to the use of the mouse-driven line tool. At the bottom-right of the figure is the Listener window for entering programming language expressions.

FIGURE 5. Chart 'n' Art main windows and palettes. The "BUS-SCHEDULE" window is instance of a kit which can hold data, pictures, and procedures for creating the diagrams in a drawing window; the "Transcript" window reveals linguistic equivalents to direct manipulation operations; and the "Listener" window is where users can experiment with expressions.



```
(set-color *Pink-Color*)
(make-oval :size '(22 16) :position '(238 96))
(set-color *Black-Color*)
(set-color *Red-Color*)
(make-line :point-1 '(125 241) :point-2 '(143 153))
(make-line :point-1 '(159 153) :point-2 '(174 81))
(make-line :point-1 '(174 81) :point-2 '(188 50))
(set-color *Light-Blue-Color*)
(make-line :point-1 '(143 153) :point-2 '(160 153))
(make-line :point-1 '(174 81) :point-2 '(176 81))
```

"Lyons"

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	

```
;       - Loading binary file "ccl;binaries:user:Chart 'n' Art:E
nd-User
;         Language.fasl"
;       - Loading binary file "ccl;binaries:user:Chart 'n'
;         Art:Chart-n-Art.fasl"
;       - Loading binary file "ccl;binaries:user:Chart 'n'
;         Art:Methods-To-Remove.fasl"
;   - Providing system chart-n-art
Welcome to Macintosh Common Lisp Version 2.0.1!
? (make-line :point-1 '(174 81) :point-2 '(176 81))
```

4.2 Self-disclosure in Chart 'n' Art

Chart 'n' Art employs self-disclosure to gently introduce its application-oriented language to the designer. The system exhibits the three essential properties from Section 3.3 for the pedagogical use of self-disclosure:

1. For every elementary mouse action which has a command language analog, the system will disclose that expression to the user.

   In CNA nearly every release of the mouse button causes a Lisp expression to appear in the Transcript window: changing colors, creating, moving, resizing gobjects, etc.

2. The system will indicate to the user groups of disclosed expressions connected with a single operation.

Chart 'n' Art prints an extra line break in the transcript window to visually group disclosed expressions connected with a single operation. Elemental operations include the simultaneous movement of a group of selected objects which appear in the transcript as a series of `move-gobject` expressions without extra line breaks.

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.

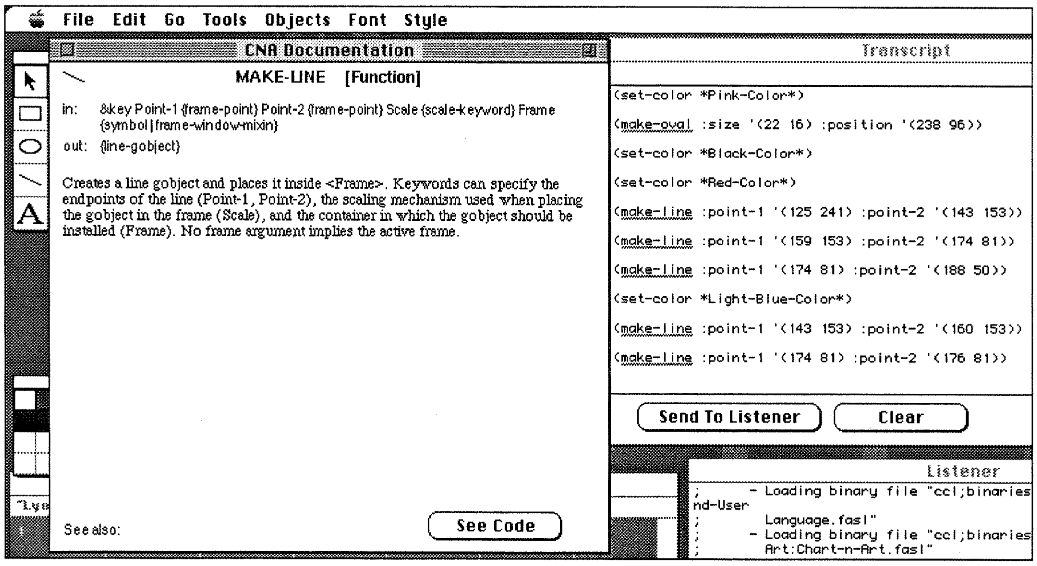
   If users undo their last direct manipulation action, then press the "Send to Listener" button in the transcript, and then press the enter key with the cursor in the Listener window, the effect is a redo. A single menu command called "Redo with Language" is planned to automate this process.

Although CNA does not yet use self-disclosure to its fullest extent, preliminary tests with Chart 'n' Art involving two subjects performing a simple drawing task indicate that non-Lisp programmers can quickly identify the basic open-parenthesis-operator-operand-close-parenthesis syntax of the language by examining items generated in the Transcript. The rest of this section describes how self-disclosure guidelines are followed in CNA.

4.2.1 Disclosures should be maximally generalizable

As illustrated by the `make-line` disclosure in the Transcript window in Figure 5, CNA attempts to aid generalizability by using Lisp keyword parameters such as `:point-1`. 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. Chart 'n' Art also tries help designers induce programming knowledge by linking disclosed function names to on-line documentation about the functions. As shown in Figure 6, designers can click the mouse on underlined function names in the Transcript window to get information about their semantics and legal parameters.

FIGURE 6. Chart 'n' Art on-line documentation for the `make-line` command. This window appeared after the user clicked on the underlined function name in the Transcript.



MAKE-LINE [Function]

in: &key Point-1 {frame-point} Point-2 {frame-point} Scale {scale-keyword} Frame {symbol|frame-window-mixin}

out: {line-gobject}

Creates a line gobject and places it inside <Frame>. Keywords can specify the endpoints of the line (Point-1, Point-2), the scaling mechanism used when placing the gobject in the frame (Scale), and the container in which the gobject should be installed (Frame). No frame argument implies the active frame.

See also:

```
(set-color *Pink-Color*)
(make-oval :size '(22 16) :position '(238 96))
(set-color *Black-Color*)
(set-color *Red-Color*)
(make-line :point-1 '(125 241) :point-2 '(143 153))
(make-line :point-1 '(159 153) :point-2 '(174 81))
(make-line :point-1 '(174 81) :point-2 '(188 50))
(set-color *Light-Blue-Color*)
(make-line :point-1 '(143 153) :point-2 '(160 153))
(make-line :point-1 '(174 81) :point-2 '(176 81))
```

```
;       - Loading binary file "ccl;binaries
nd-User
;         Language.fasl"
;       - Loading binary file "ccl;binaries
;         Art:Chart-n-Art.fasl"
```

4.2.2 The system should facilitate experimentation with disclosures

Currently, CNA users can experiment with disclosures by selecting lines in the Transcript window and pressing the "Send to Listener" button. This causes the disclosed text to appear in the interpreter's Listener window. In the Listener, designers can optionally change parameters and then press the return key to execute the Lisp expression. I plan additional mechanism to specifically facilitate the composition of functions. A new button will be added to the Transcript which 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.

4.2.3 Self-disclosure should be scaffolded

I plan to develop a method of categorizing programming expressions in terms of the knowledge required to understand them based on existing domain models of Lisp (Mastaglio, 1990) and from observing designers using programmable tools (see Section 6.0). These categories will also be used by the user model to describe an individual designer's expertise, most likely using a list of programming concepts acquired. The arbiter module will choose disclosures which require knowledge matching or slightly exceeding that of the user.

4.2.4 Disclosures should provide coverage of essential programming concepts

Chart 'n' Art disclosures currently reveal the gobject maker, mover, and resizer functions, as well as lower-level functions for manipulating groups of gobjects. I plan to use self-disclosure to introduce programming expressions for adjusting the selection and iterating over kit cells. Studies are planned (see Section 6.0) to determine the most important programming concepts and determine patterns of direct manipulation use in CNA. I will use results from these studies to seed the CNA interface in order to provide adequate coverage of essential programming expressions.

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

Chart 'n' Art users can combine direct manipulation with linguistic operations in different ways. Chart 'n' Art supports the ability to select objects with the mouse and specify the operation on those objects by typing in the Listener window. User can also combine interaction modes when making gobjects. For example, a red line is produced by selecting red from the color palette with the mouse and then typing the `make-line` expression. Future versions of the system will allow users to specify coordinate parameters to typed expression by selecting the screen location with the mouse, as is possible with AutoCAD.

4.2.6 Disclosures should be subtle and browsable

Disclosures in CNA are short Lisp expressions displayed in a small font in the compact scrollable Transcript window. If the designer has a large monitor she can move this window away from the area of design activity. I also plan to offer designers the option of shrinking the Transcript window to a minimum size. To facilitate browsing, I am also considering making thumbnail-sized before and after snapshots which appear in the Transcript window beside each group of disclosures.

4.3 Related systems

The use of self-disclosure to help introduce designers to programming is clearly influenced by a significant collection of prior work. Below are existing systems which are related to Chart 'n' Art.

SchemeChart. A predecessor to the Chart 'n' Art system is SchemeChart (Eisenberg and Fischer, 1994), a programmable tool with an application-oriented language based on Scheme. Although SchemeChart does not emphasize self-disclosure *per se,* it does offer chart designers unique learning opportunities through its "query mode." While in query mode, user selection of a component of a previously constructed chart causes the system to display a list of SchemeChart expressions that could be used to manipulate that component.

The Lisp Tutor. The Lisp Tutor (Anderson and Reiser, 1985) guides learners through a collection of problems to be solved using the Lisp language. The system relies on a sophisticated domain model of Lisp to anticipate user problems and offer suggestions. Learners can ask the tutor to fill in parts of the solution program for them and explain why these parts are necessary. Chart 'n' Art's learning opportunities are more contextualized than the Lisp Tutor; but like the Lisp Tutor, CNA will rely on a user model to constrain pedagogical feedback.

ProNet. Although not related to language instruction, ProNet (Sullivan, 1994) demonstrates the potential for learning by observation. In this system learners can induce knowledge about local-area networks while designing networks. This seemingly paradoxical situation is possible because ProNet proactively adjusts the network being built to meet predefined design criteria. Users can then ask the system why adjustments were made.

AppleScript[1] *Script Editor.* AppleScript is a general purpose language for sending high-level events to compliant Macintosh applications, which include the Finder file system interface, drawing programs, spreadsheets, and word processors. The AppleScript Script Editor is a convenient tool for composing high-level event calls. Beginning AppleScript users can learn about the language by putting the editor in record mode. Subsequent user activity in a compliant application causes code to appear in an editor window. Like AutoCAD, the record feature is mainly available for convenience, not pedagogy. The editor does not attempt to scaffold its feedback or provide coverage of essential language constructs.

Explainer. The illustrative power of programming examples is demonstrated by Redmiles' Explainer system (Redmiles, 1993). This prototype programming environment allows software designers to explore examples relevant to their current task through various "perspectives" and "views." Redmiles' work acknowledges that designers can benefit from the ability to actively investigate programming knowledge offered by the system. In CNA this is facilitated by enabling designers to experiment with disclosures. Explainer supports active exploration by automatically highlighting information from various views related to the current user selection.

1. AppleScript and Macintosh are is a registered trademarks of Apple Computer Corporation.

5.0 Assessing and refining self-disclosure

The self-disclosure guidelines presented in Section 3.4 and the current self-disclosure mechanisms in the Chart 'n' Art prototype system represent the first step towards a theoretical and technical framework for the pedagogical use of self-disclosure. Iterative assessment and refinement of this framework are necessary to achieve a valid and effective approach for introducing designers to programming. For my dissertation research, assessment will occur in two phases: formative and summative. Formative evaluations will consist of several informal observational studies designed to answer specific research questions leading to iterative refinements of the self-disclosure framework. Summative assessment will be a single formal user study designed to answer more general research questions.

5.1 Formative assessment

Formative assessment of self-disclosure will require a sufficiently functional and stable prototype on which to conduct naturalistic evaluations. This requirement motivates the need to first and foremost refine and extend the set of direct manipulation and programming operations available in CNA. I plan observational studies of designers interacting with AutoCAD, a scriptable version of an image processing tool, PhotoShop[1], and the current version of CNA to inform the next iteration of the system. Through a series of informal user studies employing this revised version of CNA, I will then begin to explore numerous alternatives to presenting disclosures and facilitating designer-interaction with them. Over the course of my dissertation there will be several opportunities for me to observe CNA used in a naturalistic setting. For example, the upcoming course "Designing Systems for Learning" will involve students using CNA to make diagrams and learn its application-oriented language. Table 3 outlines some of the specific research questions I will address during formative assessment, the evaluative methods I plan to use, and the relevance of these questions towards the self-disclosure guidelines.

5.2 Summative assessment

Similar to the evaluations used by Sullivan (Sullivan, 1994), summative assessment will be used towards the end of my dissertation work to identify the differences in the kinds of insights disclosures provide compared to other techniques. Indeed, alternative means of introducing designers to

1. PhotoShop is a registered trademark of Adobe Corporation.

programming—such as on-line help, interactive tutorials, or annotated example programs—would almost surely add value to a self-disclosing design tool. Therefore, to try to demonstrate that self-disclosure is better than other techniques would not be a very profitable assessment strategy. Instead, as shown in Table 4, I have identified general research questions relevant to issues involved in introducing designers to programming (see Section 2.3), which I will attempt to address through a formal user study of CNA.

TABLE 3. Research questions, evaluative methods, and relevance to self-disclosure guidelines.

Research question	Proposed assessment method	Generalizability	Experimentation	Scaffolding	Coverage	Combined interface	Subtle & browsable
What are the kinds of things designers want/need to do with a direct manipulation interface? With a programming language?	Observe designers using AutoCAD, PhotoShop, and CNA.				✓	✓	
What is the most generalizable form for disclosures?	Conduct informal studies of CNA designers reacting to disclosures using keyword parameters, capitalization, and optional parameters.	✓					
What order of disclosures best aids language acquisition?	Develop a partial ordering of CNA programming knowledge based on the Lisp Critic and refined using programming walkthroughs (Lewis, et al., 1990) and interviews with CNA users.			✓			
What is the appropriate level of obtrusiveness of disclosures?	Ask CNA designers to subjectively evaluate the size and placement of the Transcript window, as well as disclosures localized to the mouse position.						✓
Are the kinds of things designers want to do with disclosures supported by the system?	Observe how CNA designers use disclosures and interview them afterwards.		✓				

5.2.1 Subjects

Subjects for the summative study will be graphic arts and environmental design students enrolled at the University of Colorado. I will interview students in an attempt to select approximately 16 students with a minimal background in programming and a maximal background in design who will be randomly assigned to either a test group or a control group.

TABLE 4. Research questions, evaluative methods, and relevance to issues involved in introducing designers to programming.

Research question	Proposed assessment method	Approachability	Time for learning	Knowing the possibilities	Estimating the effort
Does disclosure increase the likelihood that designers will employ programming in their activity?	Compare the number and quality of expressions typed in Listener window between control and test groups.	✓			
Do disclosures inhibit creative flow?	Measure time taken to complete the task in control and test groups. Ask subjective flow questions in the posttest.		✓		
Do disclosures increase designers' awareness of the cost and benefits of programming?	Ask subjects in the posttest how programing might help a hypothetical task and how long they estimate a programming solution might take to devise.			✓	✓
What kinds of changes in programming knowledge occur because of self-disclosure?	Compare programming knowledge scores between pretest and posttest for both groups.				
Are designers motivated to learn more about the language?	Ask subjective questions in the posttest about motivation levels.				

5.2.2 Design

The study will be cross-sectional, with each subject participating once in a single three-phase trial involving a programming and design pretest, an information graphics design task using CNA, and a programming and design posttest. Members of the test group will receive disclosures from CNA while they use the system, while members of the control group will not see disclosures but they have access to a printed CNA manual.

The pretest will assess subjects' backgrounds in design and programming. The design task will be a non-trivial information graphics design problem that does not require programming and has a number of solutions. All subjects will be asked to work on the task until they feel they have a design which best portrays the given data. CNA will be instrumented so that interactions with the direct manipulation and linguistic interfaces will be recorded, supplemented by periodic screen snapshots. The posttest will assess changes in subjects' programming knowledge, attitudes towards programming, and abilities to estimate the costs and benefits of programming, among other things.

5.3 Outline of proposed work

Although work on the self-disclosure guidelines and a prototype self-disclosing tool has already begun, I have yet to initiate the bulk of my dissertation efforts, including identifying appropriate operations for information graphics designers, thoroughly applying the self-disclosure guidelines to CNA, and using this system to evaluate the effectiveness of self-disclosure. The key remaining research efforts (and the semester during which they will take place) consist of

1. observing designers using commercially available programmable design tools as well as CNA (Spring 1995),
2. refining and extending CNA direct manipulation and programming functionality (Spring - Summer 1995),
3. iteratively assessing and refining self-disclosure guidelines (Spring - Fall 1995), and
4. conducting a formal user study of CNA (Fall 1995 - Spring 1996).

6.0 Summary

My proposed dissertation research is significant because it explores mechanisms for empowering designers with the skills to begin programming their tools, thus providing them with an expressive medium previously reserved for computer experts. Results from my research will, if successful, provide a model for embedding end-user language learning opportunities into programmable applications. My work will also provide developers with valuable insight into the coordinated design of their tool's direct manipulation and programming interfaces.

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 which revealed presentation techniques while they chose from high-level charting operations. Self-disclosure may prove to be a powerful and general technique for presenting new knowledge to users performing any number of computer-based tasks.

References

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

Anderson, J.R., and B.J. Reiser (1985). *The LISP Tutor.*

Brown, Ann L., Doris Ash, Martha Rutherford, Kathryn Nakagawa, Ann Gordon, and Joseph C. Campione (1993). Distributed expertise in the classroom. In *Distributed Cognitions.* Edited by G. Salomon. Cambridge, England: Cambridge University Press.

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

Burton, R.R., J.S. Brown, and G. Fischer (1984). Analysis of Skiing as a Success Model of Instruction: Manipulating the Learning Environment to Enhance Skill Acquisition. In *Everyday Cognition: Its Development in Social Context.* Edited by J. L. B. Rogoff. 139-150. Cambridge, MA - London: Harvard University Press.

Carroll, J.M., and M.B. Rosson (1986). *Paradox of the Active User.* RC 11638.

Collins, A.M., J.S. Brown, and S.E. Newman (1989). Cognitive Apprenticeship: Teaching the Crafts of Reading, Writing, and Mathematics. In *Knowing, Learning, and Instruction.* Edited by L. B. Resnick. 453-494. Hillsdale, NJ: Lawrence Erlbaum Associates.

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience.* Harper-Collins Publishers.

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

Derlega, Valerian J., and John H. Berg (1987). *Self-Disclosure.* Theory, Research, and Therapy. New York: Plenum Press.

DiGiano, Chris, and Mike Eisenberg (submitted 6 October 1994). Supporting the end-user programmer as a lifelong learner. *Proceedings of the National Educational Computing Conference.*

Eisenberg, M. (1991). *Programmable Applications: Interpreter Meets Interface.* Department of Electrical Engineering and Computer Science, MIT. 1325.

Eisenberg, M., and G. Fischer (1994). Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance. In *Human Factors in Computing Systems, CHI'94 Conference Proceedings (Boston, MA).* 431-437.

Elsom-Cook, Mark (1990). *Guided discovery tutoring: a framework for ICAI research.* London: Paul Chapman Publishing.

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, G. (1991a). The Importance of Models in Making Complex Systems Comprehensible. *Proceedings of the Mental Models and Human Computer Communication: Proceedings of the 8th Interdisciplinary Workshop on Informatics and Psychology*, 3-36.

Fischer, G. (1991b). Supporting Learning on Demand with Design Environments. In *Proceedings of the International Conference on the Learning Sciences 1991 (Evanston, IL)*. Edited by L. Birnbaum. 165-172. Charlottesville, VA: Association for the Advancement of Computing in Education.

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.

Gellenbeck, Edward M., and Curtis R. Cook (1991a). Does Signaling Help Professional Programmers Read and Understand Computer Programs? In *Empirical Studies of Programmers: Fourth Workshop*. 82-98.

Gellenbeck, Edward M., and Curtis R. Cook (1991b). An Investigation of Procedure and Variable Names as Beacons during Program Comprehension. In *Empirical Studies of Programmers: Fourth Workshop*. 65-81.

Gleitman, Lila R. (1984). The current status of the motherese hypothesis. *Journal of Child Language* 11 (1): 43-77.

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.

Hayes-Roth, Frederick (1978). Learning by example. In *Cognitive Psychology and Instruction*. 27-38. New York: Plenum Press.

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

Lewis, Clayton, and Gary M. Olson (1987). Can Principles of Cognition Lower the Barriers to Programming? In *Empirical Studies of Programmers: Second Workshop*. 248-263.

Lewis, C.H., P. Polson, C. Wharton, and J. Rieman (1990). Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces. In *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*. 235-242. New York:

Locke, John (1964). *An Essay Concerning Human Understanding.* Cleveland: Meridian Books (originally published in 1690).

Mackay, Wendy E. (1991). Triggers and Barriers to Customizing Software. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*. 153-160.

Mackinlay, Jock (1986). Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics* 5 (2): 110-141.

Mastaglio, Thomas W. (1990). A User-Modelling Approach to Computer-Based Critiquing. PhD Dissertation, University of Colorado.

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.

Redmiles, D.F. (1993). Reducing the Variability of Programmers' Performance Through Explained Examples. In *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*. 67-73.

Resnick, L.B. (1989). Introduction. In *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser.* Hillsdale, NJ: Lawrence Erlbaum Associates.

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

Scardamalia, Marlene, and Carl Bereiter (1991). Higher levels of agency for children in knowledge building: a challenge for the design of new knowledge media. *Journal of Learning Sciences* 1 (1): 37-68.

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

Simon, H.A. (1981). *The Sciences of the Artificial.* Cambridge, MA: The MIT Press.

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

Sullivan, Jim (1994). ProNet: A proactive design environment. PhD Dissertation, University of Colorado.

Tufte, Edward R. (1990). *Envisioning Information.* Cheshire, Connecticut: Graphics Press.

Venolia, Dan (1994). T-Cube: A fast, self-disclosing pen-based alphabet. *Proceedings of the Computer-Human Interaction Conference*, 265-70.

Vygotsky, L. S. (1978). *Mind in Society.* Cambridge, MA: Harvard University Press.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems.* Los Altos, CA: Morgan Kaufmann Publishers.