Self-Disclosing Design Tools: An Incremental Approach Toward End-User Programming

Christopher John DiGiano

CU-CS-822-96

SELF-DISCLOSING DESIGN TOOLS: AN INCREMENTAL APPROACH TOWARD END-USER PROGRAMMING

CHRISTOPHER JOHN DiGIANO

University of Colorado at Boulder

Abstract

The typical direct manipulation software application currently suffers from a lack of expressiveness in that users are limited to the functionality anticipated by the system's graphical interface. A *programmable application* provides users with the ability to transcend the constraints imposed by its direct manipulation interface through end-user programming. By weaving programming expressions into their regular interaction with such tools, users can potentially accomplish tasks faster and more accurately, and take on whole new kinds of problems they had never considered previously. However, for many users the transition from a direct-manipulation style of interaction to a more formal, language-based one can be a daunting challenge.

This research seeks to address the language learnability problem inherent in programmable design tools by embedding "self-disclosure" mechanisms into these systems. Self-disclosing design tools can—as they are being used—present short, relevant programming language examples. Through exposure to these incremental language learning opportunities within the context of authentic activity, this work hypothesizes that users can acquire end-user programming skills with minimal distraction from the tasks at hand. To test this theory, the self-disclosing drawing tool *Chart 'n' Art* was developed and evaluated in a series of studies. The final analysis reveals that, at least for some individuals, self-disclosing tools can have a positive educational impact on programming language acquisition with only limited interference to user activity, and that the disclosures themselves can facilitate the composition of programming expressions.

Acknowledgments

This report is essentially a reformatting of my Ph.D. dissertation which was completed Dececember 6, 1996. I am indebted to a number of people and organizations who made my thesis work possible. First, I should recognize the National Science Foundation (grants RED-9253425, REC-9553371, REC-9631396) for constant financial support and the CU Department of Computer Science and Institute of Cognitive Science for travel funding. I would also like to thank Jim Ambach, Hal Eden, Gerhard Fischer, Corrina Perrone, Alex Repenning, and others in the Center for LifeLong Learning & Design for their helpful feedback on this work, as well as collaborators at Apple Computer, Inc.—Jim Spohrer and Jamie Dinkelacker to name a few—who always had useful comments. I am particularly grateful for the opportunity to interact closely with my advisor, Mike Eisenberg, whose creative work provided the inspiration for my becoming a better writer, speaker, and researcher. The dissertation also benefited from the wisdom and experience of Clayton Lewis with whom I frequently met. The rest of my thesis committee, Tom Landauer, Jim Martin, and Nancy Butler Songer, also provided invaluable comments and advice.

Additional thanks are in order for those at the Boulder Daily Camera, CU Publications Department, and the University of Michigan's Collaboratory for Research on Electronic Work who provided insight into the process of information graphics design. I also appreciated working with Stan James and Susanne Commisso on their related student research projects. I would especially like to thank Eric Blough and Francesca Iovine for the countless hours they devoted to helping organize the user studies and other aspects of this research. The studies would also not have been possible without the assistance of Halden Field, Stefanie Lindstaedt, Anders Mørch, Ann Nishioka, Cyndi Rader, Markus Stolze, and others. Finally, I am grateful for the chance to collaborate with John Rieman in designing the summative assessment.

Table of Contents

List of Tables

List of Figures

1. Introduction

...it is possible to design computers so that learning to communicate with them can be a natural process, more like learning French by living in France than like trying to learn it through the unnatural process of American foreign-language instruction in classrooms.

(Papert, 1993, p. 6)

1.1 Overview

It is no surprise that much of design—from architecture to fashion to music—is computer assisted. What is surprising, at least for technologists, is how little processing power is actually exploited to create these artifacts. In many cases, the nature of the activity—the actual steps in the design process—has changed very little since the advent of computers. Although computers accelerate some of these steps, rarely do they completely eliminate the tedious parts of design and free designers to do more creative work.

End-user programming offers some hope—a means for shifting uninteresting responsibilities from designer to machine. Programming can empower designers with the ability to transcend the limitations of their software and attempt completely new kinds of designs that were simply impractical before. The problem is that designers are not programmers.

Or are they? This report explores the possibility that end users can be introduced to programming languages *while designing*, through a technique called *self-disclosure.* This research hypothesizes that language learning opportunities can be integrated into the experience of using computational tools in a way that is naturally situated and minimizes disruption of design "flow" (Csikszentmihalyi, 1990). This notion of embedded learning opportunities is illustrated and evaluated through a prototype self-disclosing programmable drawing tool called *Chart 'n' Art.*

1.2 The Complexity Crisis

For many complex tasks, computer-based tools have arguably failed to live up to their potential to revolutionize design. In many such cases, designers of complicated artifacts have rejected domain-oriented software such as charting packages in favor of domain-independent tools such as generic drawing programs. Using such tools, the design process can often be laborious and inaccurate, as the case studies appearing in Chapter 2 will make clear. But designers are willing to tolerate this loss of productivity and accuracy, this research claims, because by using low-level generic tools they maintain expressive control over their work. Furthermore, the generic tools traditionally present a simple direct-manipulation interface that is familiar to designers, especially those with experience in physical media such as pencil and paper.

The response of the software industry to the perceived rigidity of domain-specific tools and the tedium of using domain-independent tools has been to blur their distinction. For instance, modern charting packages such as DeltaGraph Pro (DeltaPoint Inc., 1996) now feature drawing tools for adding custom chart elements and adjusting colors, line weights, and fonts. At the same time, low-level design programs now sport a growing list of specialized functionality, for example—in the case of drawing programs such as

Illustrator (Adobe Systems Inc., 1996a)—commands for making pie charts, three-dimensional shapes, or spirals. Not only would these added features seem to make domain-dependent software more flexible and domain-independent software more powerful, but the additional functionality provides software developers with key selling points in an increasingly competitive market. The problem is that this "creeping featurism" had led to a burgeoning complexity in design software that renders such tools more and more difficult to learn to use. The result is a complexity crisis: in an effort to help designers of complex artifacts make better use of computational tools, software developers have paradoxically created software behemoths that are themselves complex.

End-user programming has emerged as a possible means for mitigating the complexity crisis. Proponents claim that programming can aid designers by automating tasks, by enabling them to extend the functionality of limited applications, and by providing a symbol system that enables users to think more productively and possibly more creatively about their activity (c.f Eisenberg, 1995; Nardi, 1993; Papert, 1993). Since programming offers the potential for designers to personally modify or enhance the feature sets of their software, the implication is that developers of the underlying design tools can return to publishing simpler software with base-level functionality. However, end-user programming introduces its own source of complexity in that users must somehow learn to specify their intent using a particular formal language. Whether the end-user language is graphical or textual, designers must learn the vocabulary, syntax, and semantics of a notational system.

The remainder of this chapter outlines the complexity crisis from the perspective of particular field of design, *information graphics*, and offers background information on this specialty that will help in understanding discussions to follow. Although the focus is on the field of information graphics, the themes echoed throughout this work have much broader implications on the design of *learnable* high-functionality systems to support a wide range of complex and creative tasks. This issue of learnability will only become more urgent as the number and variety of these types of tools increase.

1.3 Custom Information Graphics

Information graphics is the design of charts, graphs, and other diagrams used to portray data. Examples of standard types of information graphics include the column chart, pie chart, line chart, and scatter plot. These diagrams are characterized by a collection of chart elements such as bars, wedges, lines, or points whose attributes correspond to one or more series of data. Attributes can include size, position, color, weight, orientation, etc. This characterization describes other forms of information graphics, not just the standard types. Consider the famous diagram in Figure 1-1 of Napoleon's Russian campaign of 1812. The main chart element in this case is the band that generally flows from left to right and back. The position of the band corresponds to locations visited by Napoleon's troops on a map of Russia. The width of the band represents the number of troops. Finally, the color of the band corresponds to the direction of the army's movement, gray representing Napoleon's advance and black his retreat.

What is remarkable about the illustration of Napoleon's march is the way it manages to represent many dimensions of data with a single compact diagram. This choice of mappings between data and graph elements is uniquely suited for the information being illustrated. Such "custom" information graphics are found in a variety of media from scientific journals to business reports to advertisements. Chapter 2 describes two

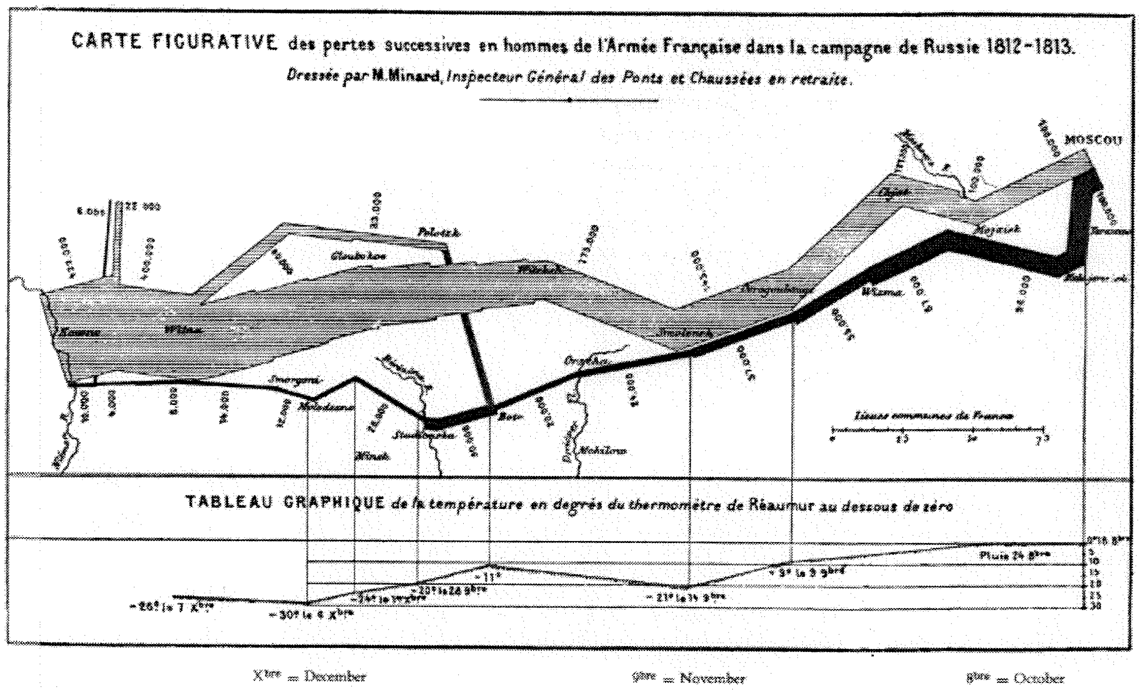


Figure 1-1. Napoleon's Russian campaign of 1812, originally drawn by Charles Joseph Minard in 1861. Reproduced from (Tufte, 1983, p. 41).

instances: The first, a unique temperature chart, is from a daily newspaper, and the other, a representation of meeting activity, is from a scientific article about Computer Supported Cooperative Work (CSCW).

This report also uses the term "custom information graphics" to describe diagrams that use a standard data mapping, but are stylized or embellished in some way to distinguish them from the typical output of computer-based charting tools. Chapter 2 describes two such cases, one involving an elongated column chart and the other a fancy pie chart.

1.4 Designers of Information Graphics

The above examples of custom information graphics serve to highlight the variety of designers involved in the production of such diagrams. In the case of the temperature chart, the designers are professional graphic artists employed by the newspaper. In the case of the meeting chart, the designers are computer scientists and psychologists. A rough categorization of designers pertinent to this research consists of the following groups:

- ***professional graphic designers*** who are hired by numerous clients to create a variety of illustrations as a full-time job,
- ***data-owners*** such as scientists, statisticians, and business people who not only produce the diagrams, but are responsible for actually collecting or generating the data, and
- ***"gardeners"*** (Gantt & Nardi, 1992) employed by a particular investigative or business team to help with the more advanced computing tasks.

Each of these classes of individuals comes to the task of design with a different set of skills and constraints. Professional graphic designers typically bring considerable experience with activities involving hand-eye coordination, while data-owners often have significant mathematical background. Gardeners usually have some background in domain of the data being graphed, but have a special technical role because of their more advanced computing skills, possibly including programming. In order to be successful, graphic artists typically must not only work creatively, but also efficiently, as they often juggle a variety of projects simultaneously. The primary job of data-owners is to do science or conduct business, not generate charts, but on the other hand they have a vested interest in their graphics communicating information effectively. Like graphic artists, gardeners produce a large number of charts, but as part of research or business teams, the types of data they graph may be more consistent.

1.5 Tools for Information Graphics

Ever since the teletype printer, computers have been used to plot charts and graphs. The specialized programs written to produce these diagrams have evolved into present-day charting tools such as DeltaGraph Pro which provide high-level functionality for automatically producing standard information graphics. Although computational support for charting has existed for some time, it is only recently (in the last few decades by some estimates) that professional graphic designers have abandoned their X-Acto™ knives and contact paperand turned to computers to help produce their diagrams.

Curiously, among many professional artists and other designers the preferred method for creating information graphics with computers involves many of the same steps as it did without them. Instead of taking advantage of high-level charting tools such as DeltaGraph, these designers rely on lower-level drawing packages such as MacDraw (Claris Corp., 1996) that allow the manipulation of primitive shapes such as rectangles, lines, and ovals. A plausible explanation for this phenomenon is that the high-level tools simply do not support the kinds of customizations desired by designers. For example, a designer wishing to create a graphic with a unique data mapping such as in the Napoleon's march diagram would today have little choice but to resort to a drawing tool. Furthermore, a designer creating a stylization of a standard chart might also prefer to produce the entire design in a flexible and familiar drawing tool rather than learn chart modification tricks in a charting tool or techniques for exporting charts from a graphing package into a drawing program.

In addition to charting and drawing tools, a third class of information graphics software that is only now becoming practical is the programmable drawing tool which combines mouse-based painting and drawing features with an end-user programming language. One of the best known examples of programmable drawing tools is the drafting software AutoCAD (AutoDesk Inc., 1996) which integrates a command language with a point-and-click interface. Programmable drawing tools such as AutoCAD are part of larger class of systems known as "programmable applications" (Eisenberg, 1995) which meld a direct manipulation interface with a programming language, typically an interpreted textual language such as Lisp or BASIC. A programming language embedded in a drawing tool can empower designers with the ability to create illustrations more quickly and more accurately than using a drawing program but without constraining the design space the way standard charting packages tend to do. With a programming language, designers can in fact explore whole new kinds of diagrams that make use of the language's ability to iteratively perform complex computations or drawing operations. One such example is the trapezoidal bar chart

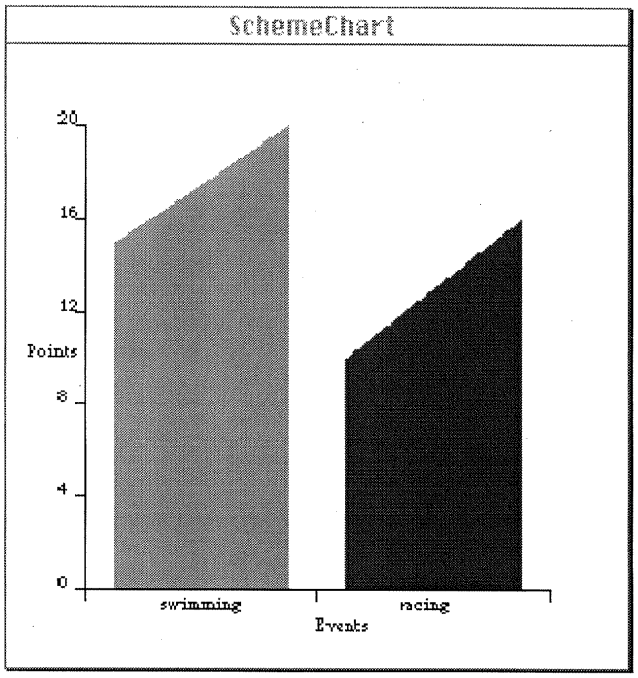


Figure 1-2. A custom information graphic created by the programmable drawing tool, SchemeChart. The unique "trapezoidal" bar chart is useful for depicting maximum and minimum range values as opposed to the single data values denoted by standard bar chart levels.

shown in Figure 1-2 which was generated with a prototype programmable drawing tool, SchemeChart (Eisenberg & Fischer, 1994).

Although most commercially available examples of programmable drawing tools are limited to rather specialized domains such as drafting, circuit design, and geometric analysis (Jackiw & Finzer, 1993), laboratory prototypes such as SchemeChart are emerging which can support more general forms of information graphics. A number of these will be describe in later chapters. At the same time, there is a software industry trend toward integrating language features into productivity software, as evidenced by the proliferation of Visual BASIC (Microsoft Corp., 1996b) in Microsoft's recent products. The implication is that popular drawing programs such as Adobe Illustrator or FreeHand (Macromedia Inc., 1996b) may one day be augmented with programming languages.

1.6 Design Tool Tradeoffs

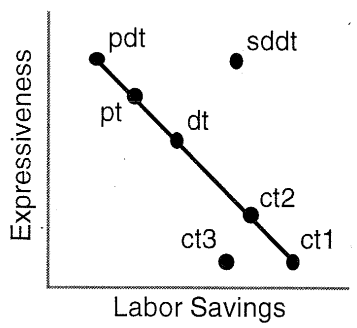
The above description of programmable drawing packages suggests that with their added flexibility and labor-saving automation such tools would be ideal for custom information graphics. However, one must not leave out a critical factor in the labor savings estimate for programmable design tools: training. Although programming solutions to many custom information graphics design tasks might seem simple and elegant to the computer scientists developing prototype programmable drawing tools, these language-based techniques must be learned by the designer. Learning the specific language of a programmable drawing tool and understanding how to integrate it into the tool's direct manipulation facilities could be a daunting and time consuming task. Even for designers with programming experience, such as many "gardeners," the language of a programmable application is rarely completely familiar. Indeed, experienced programmers who begin using system based on a common language such as Lisp or BASIC must still acquire the tool's "domain-enriched" dialect for specifically supporting design activity.

The choice of information graphics tools mentioned so far—charting tools, drawing tools, and programmable drawing tools—represent a tradeoff between expressiveness and labor savings as illustrated by points ct1, dt, and pdt in Figure 1-3. The designer must decide whether to use more flexible but typically more time consuming tools to create custom artifacts, or be satisfied with the automatic output of a charting tool. The trend indicated by the diagonal line is that the more expressive the program, the more effort it requires to use. Note how other types of tools also fit the trend such as the advanced charting tool (ct2) with added features for editing automatically created charts (e.g. by changing the color of graph elements or the default position of axis labels or legends.) These systems typically require more effort from the designer who must learn to navigate the additional menus and dialog boxes that control these options. Pixel-based painting programs such as Photoshop (Adobe Systems Inc., 1996c) represent another point, pt, along the trend line. These tools allow users to make even finer grained customizations to charts and graphs than drawing tools, e.g. by smudging the tops of bars in a bar graph to indicate uncertainty in a data set. However, the price for this added flexibility is the tedium of delicately operating pixel-level tools such as an on-screen airbrush.

Clearly, the expressiveness-labor savings diagram is an oversimplification of the factors influencing the adoption of specific design tools. Designers' prior experience with various tools and technologies will undoubtedly play a role. For example, a designer with no spreadsheet experience might consider a spreadsheet-based charting tool such as Microsoft Excel (Microsoft Corp., 1996a) to be more effortful than a designer who had used spreadsheets, and place the system at point ct3 in the diagram. In the case of programmable drawing tools, the perception of labor savings will obviously depend on programming background. The diagram also does not take into account the long term benefits of adopting various design strategies. For instance, a programmable drawing tool might have a high initial cost, but the long term labor savings might eventually place it further to the right in Figure 1-3. Studies of software adoption in other domains (Mackay, 1991; Rieman, in press), however, suggest that users generally seek short term solutions to the task at hand and only learn about new software opportunistically.

1.7 Self-Disclosing Programmable Design Tools

This report proposes an alternative form of graphics software called the *self-disclosing programmable design tool.* A self-disclosing design tool is a programmable application with embedded language learning opportunities designed to gently introduce designers to programming. The goal, indicated by point sddt in Figure 1-3, is an expressive tool perceived by designers to require much less effort to learn than typical programmable tools. A self-disclosing design tool acts like standard direct manipulation software with one important exception: Mouse actions can trigger the presentation of one or more short, instructive, programming language examples directly related to the user's current activity. In this way, by revealing connections between direction manipulation interaction and programming expressions, such a tool has the potential of progressively orienting the user to the language.



Key	
ct	Charting Tools
dt	Drawing Tools
pdt	Programmable Drawing Tools
pt	Painting Tools
sddt	Self-Disclosing Design Tools

Figure 1-3. Tradeoffs between expressiveness and labor savings for various computational design tools.

To better understand self-disclosure it helps to be familiar with the inspiration for this technique: human language acquisition. Although there is still considerable debate about how much linguistic knowledge is innate, most researchers agree that a necessary condition for first language development is immersion in a language-rich environment. Evidence for this basic tenet comes from "Wild Child" studies of people who as infants lived in language impoverished environments and failed to develop normal language skills (Fromkin, Krashen, Curtiss, Rigler & Rigler, 1985). A more controversial issue is the importance of the structure of the language input received by children acquiring a first language. Some linguists believe the shorter, simpler, and more grammatical expressions in child-directed speech or "motherese" is necessary for language acquisition (Gleitman, 1984). But other linguists point to cultures that have normal language abilities, yet its members speak very little directly to their children (c.f. Snow, 1986). One thing is clear, however: Caregivers rarely, if ever, offer children explicit grammar lessons. Rather, human language is somehow acquired *implicitly* from exposure to other speakers.

It is this incidental learning of spoken language that is the model for the self-disclosure technique. Self-disclosure provides a means for immersing computer users in a programming-language-rich computational environment, analogous to the conversational environment that typically surrounds human language learners. The idea is that by enabling users to observe a variety of disclosed programming language examples, they can begin to understand the vocabulary, syntax, and semantics of a programming language by analyzing and generalizing from the various expressions. Research on human learning suggests that "analysis-based generalization" (Lewis, 1988) is in fact a reasonable way for people to acquire the declarative and procedural knowledge necessary for programming (Pirolli & Anderson, 1985; Wiedenbeck, 1989).

Obviously, programming language learning is not perfectly analogous with human language acquisition. For example, programming languages are traditionally written languages, while the above discussion has focused on *spoken* language acquisition. To further refine the notion of self-disclosure, then, this project has attempted to incorporate lessons learned from various forms of learner-centered educational computing including constructionism, cognitive apprenticeship, learning on demand, and minimalist instruction. The combination of research from educational computing and language acquisition has influenced the development of a conceptual framework for the design of pedagogically effective disclosure mechanisms. This framework takes the form of the following six dimensions to be described in more detail in later chapters:

- ***Adaptivity***: the capacity of a self-disclosing system to change the content and form of disclosures to suit the needs of each user individually.
- ***Explorability***: the degree to which a self-disclosing system supports active user inquiry into the meaning and utility of disclosures.
- ***Intrusiveness***: a measure of how distracting disclosures are for users.
- ***Relevance***: the degree to which disclosures are contextualized to user activity.
- ***Scope***: the extent of hidden information that is disclosed to users.
- ***Structure***: a measure of the strictness of sequence in the presentation of disclosures to users.

1.8 Chart 'n' Art

Chart 'n' Art is a programmable information graphics tool designed as a test bed for self-disclosure mechanisms falling along various points in the six dimensions of the disclosure design space. On its surface, Chart 'n' Art looks like a standard direct manipulation drawing package such as MacDraw. In one on-screen palette, the system features a standard collection of primitive shapes—rectangle, oval, line, etc.—that can be drawn onto a window using the mouse. Another palette maintains a range of colors that can be applied to the shapes. Various menu commands enable users to cut, copy, paste, duplicate, and rearrange shapes.

Once users start drawing with Chart 'n' Art, however, they begin to notice some important differences. After almost every mouse click or drag, in addition to the expected drawing program effect the system also presents a small amount of text either by the cursor or in a special window labeled "Language." This feedback is Chart 'n' Art's way of incrementally *disclosing* its programming language to the user. In all, Chart 'n' Art offers four different disclosing mechanisms: mouse disclosures, selection labeling, alternatives disclosures, and possibility disclosures. These mechanisms, described in detail in Chapter 4, are designed to expose designers to numerous expressions of varying complexity.

Because of the general-purpose architecture of the system, Chart 'n' Art can actually disclose and respond to a variety of programming languages. Currently, two language are supported: Lisp and SK8Script. SK8Script is a prototype-based object-oriented language similar to AppleScript (Apple Computer Inc., 1993) and HyperTalk (Apple Computer Inc., 1996a) which is being developed at Apple Computer, Inc. Although Lisp was used for preliminary testing of Chart 'n' Art, it was SK8Script that was used for the final evaluation, because of its extremely readable syntax.

1.9 Evaluation

Throughout the process of creating the Chart 'n' Art prototype, a number of evaluations were conducted. These include a feasibility study, a field study of professional graphic designers, and a preliminary laboratory experiment with a variety of users. The most recent evaluation was a summative assessment of educational impact which attempted to quantify and characterize the nature of programming learning resulting from disclosures. In particular, this experiment sought to address three critical questions:

1. ***The incidental learning question.*** What can users learn about an end-user programming language simply from exposure to disclosures, without actually employing them to execute expressions?

2. ***The composition question.*** How much can users learn about writing their own expressions and especially about composing nested or sequenced statements when exposed to short, isolated disclosures as typically found in Chart 'n' Art?

3. ***The programming experience question.*** What influence does programming background have on what can be learned from disclosures in Chart 'n' Art?

1.10 Reader's Guide

The rest of this document expands on the arguments made in this introductory chapter, explains in depth the theory behind self-disclosure, describes the Chart 'n' Art prototype, and reports on the system's evaluation. Chapter 2 recounts a number of challenging information design tasks faced by real-life designers and sketches some potential solutions to these tasks involving programming. Chapter 3 then presents self-disclosure as a means of introducing these designers to programming and as a general mechanism for revealing hidden functionality in computer systems. It is here that the six dimensions of pedagogical disclosures are described in depth. Chapter 4 presents Chart 'n' Art and its various disclosure mechanisms. Chapter 5 analyzes Chart 'n' Art in terms of its evolution over a number of prototypes and its relationship both to the disclosures dimensions and to other programmable drawing tools and computer-based learning environments. Chapter 6 describes the lessons learned from three different evaluations of Chart 'n' Art with emphasis on the summative assessment involving some 30 subjects. Finally, Chapter 7 concludes by identifying limitations with Chart 'n' Art's approaches to self-disclosure and describes future work for addressing some of these limitations.

2. Case Studies of Design and Programming

To motivate the need for designers to learn programming, this chapter presents a number of real-life case studies of custom chart creation. It begins by recounting a field study of graphic artists involved in creating, among other things, custom charts. The study reveals a potential role for programming, but also a number of challenges in introducing programming into such a setting. To complement these findings three other case studies are presented, illustrating the variety of design settings that could benefit from programming and the complexity of the end-user programming learnability problem.

2.1 CU Publications Department

In the spring of 1995, a field study of professional graphic designers at the University of Colorado's Publications Department (CU Publications) was conducted to help assess the role for programming in the creation of custom charts and graphs.[1] The field study involved seven approximately two-hour visits to the publications department. The primary informants were two professional graphic artists, Designer A and Designer B, with many years of design experience and three and seven years of computer experience, respectively.

The designers at CU Publications served numerous university clients with a variety of needs including brochures, maps, magazines, and posters. For both of the informants chart creation was a relatively small proportion of their regular activity and would often be a component of a larger project such as an annual report. When they did receive a charting task the client would often provide them with an existing graph to "beautify"—that is, to make more attractive or to arrange in a more creative way that fit in the surrounding context for the graph. The software used by Designers A and B was primarily general purpose drawing and layout software, including Adobe Illustrator, Macromedia FreeHand, Adobe Photoshop, QuarkXPress (Quark Inc., 1996), and Aldus Page Maker (Adobe Systems Inc., 1996b).

The first visits were spent reviewing the designers' portfolios of charts, graphs, and other illustrations. They described the processes involved in creating the artifacts and for some designs reenacted the steps they took using the computer. Another visit was spent watching Designer A create a complicated voice-dialog diagram for an automated phone service using a combination of Illustrator and PageMaker. For the next two sessions, the informants agreed to follow a tutorial found in one of Illustrator's printed manuals that reviewed the use of the system's charting component. This component provided a simple spreadsheet for entering data which could then be mapped to a small number of standard chart types. The purpose of these sessions was to observe how the subjects reacted to the lessons provided in the manual and to elicit their opinions of Illustrator's charting functionality. In the final two sessions participants were asked follow-up questions formulated after reviewing data from previous sessions. Data collected from the field study provided valuable insight into designers' tasks, tools, and learning opportunities.

---

[1] The field study was a joint project of the author and fellow student Susanne Commisso.

2.1.1 A Custom Column Chart

One of the artifacts that Designer B described from his portfolio was the Colorado Foundation 1993 Annual Report. The report contained three simple yet elegant column charts, one of which is reproduced in Figure 2-1. As best as Designer B could remember, he produced each of these charts using the layout program QuarkXPress as follows: First, he installed a "guide" on QuarkXPress's vertical ruler which appeared as a horizontal line across the screen to mark the horizontal axis and the bottom of each column in the chart. For each data point, Designer B then installed additional guides, the position of each guide proportional to the target height of a column in the chart. The position was determined either by estimating its appropriate height relative to other guides or by performing a scaling calculation using pencil and paper or a calculator. Next, he dragged out the shape of a new rectangle and positioned and sized it so that its bottom matched the horizontal axis guide and its top matched the corresponding guide for its height. This rectangle served as a model for all the others which would share all the same attributes except height. For each of the other six columns in the chart, Designer B duplicated this model rectangle, moved the copy into position beside the other columns and then adjusted its height to match its corresponding guide.

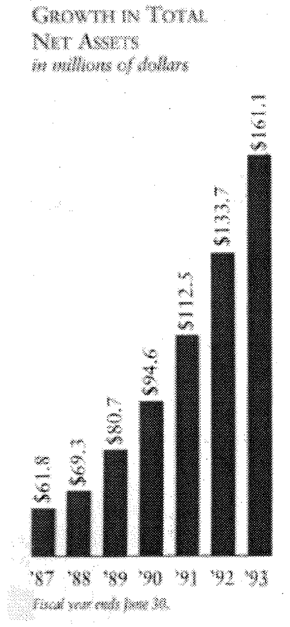


Figure 2-1. One of the three column charts Designer B created for the Colorado Foundation 1993 Annual Report.

Designer B estimated that this type of column chart took him two hours to complete. It is interesting to note that Designer B had at one time tried using automatic means for producing such charts more quickly—at least one such attempt was with Illustrator's built in charting component. In the end, he abandoned this approach and chose to stick with more familiar general-purpose drawing tools. One reason was clearly his discomfort with its spreadsheet interface for entering data values. Designer B commented that with the charting component it was "hard to figure out what goes where" and he "couldn't figure out the spreadsheet deal."

Another contributing factor to Designer B's disinterest in automatically generated charts may be a perception that such tools limit his creativity. For instance, the designer mentioned that for the annual report he wanted long, narrow columns with a certain amount of white space in between and expressed skepticism that such a custom look could be achieved with a charting tool. Designer B also commented that rather than scanning in and modifying a client's initial concept he preferred to start from scratch. Similar sentiments were echoed by Designer A who complained that Illustrator's charting component was "not creative" and made graphs that were "just like everybody else's."

Walking the designers through the Illustrator tutorial did little to change the designers' opinion of the system's charting component, it would seem. Both had difficulty entering values in the spreadsheet cells and navigating from one cell to the next. Not surprisingly, they were impressed with the component's speed in generating standard charts. But, they were also quick to critique elements of the automatically produced designs, such as the default placement of text labels. They also soon identified capabilities beyond the component's ability, such as remembering custom adjustments after the data for a chart had changed and its graph was automatically regenerated.

2.1.2 Potential for Programming

Proponents of end-user programming might argue that what is needed is a computer programming language that could help automate the chart creation process and make it more accurate without compromising designers' expressiveness. But what form of language-based support would be appropriate in this setting? A partial solution that would leverage off of Designer B's existing expertise would be if he could specify the exact height of each column by typing instead of dragging. The ability to type in a shape's attributes is now supported by more and more drawing programs including QuarkXPress. This might indeed help Designer B's particular dilemma for the narrow column chart task, but it is not a very general solution to Designer B's tedium. For instance, next week he may need to create a different chart and wish to type in the width of a shape, or perhaps the size of its border or the exact RGB values of its color. Furthermore, some scaling may be necessary to convert the input values to the actual measurements in inches, picas, or pixels used by the drawing program.

A more general solution would be for Quark to integrate a programming language into QuarkXPress.[2] Each shape could have any number of named attributes that could be changed by typing a short attribute-setting command such as

```
set the height of Rectangle 1 in DrawWindow to 61.8
```

into an interpreter window. Quark would not be saddled with having to create interface widgets for adjusting each property, and Designer B would be free to enter exact values for any aspect of a shape so long as it had a name. Scaling could be accomplished by nesting a multiplication expression and possibly an addition expression within the attribute-setting command:

```
set the height of Rectangle 1 in DrawWindow to 61.8 * 10 + 5
```

At this point, one must question how much tedium is actually being saved when such an expression must be typed for each column in the chart. But if QuarkXPress were to feature a full-fledged programming language, Designer B would also be able to create procedures for abstracting functionality and applying it to input values:

[2] All programming solutions in this chapter are presented in SK8Script, the primary language supported by the prototype self-disclosing tool, Chart 'n' Art. SK8Script is an object-oriented, English-like language related to HyperTalk and AppleScript. More information about SK8Script can be found in Chapter 4 which describes Chart 'n' Art in some detail.

```
on growColumn n, h
   set the height of Rectangle n in DrawWindow to h * 10 + 5
end growColumn
```

The designer could then type a—now much shorter—`growColumn` command for each rectangle he wished to precisely size. To simplify things even further, one can imagine that `growColumn` could be associated by Designer B with his collection of rectangles for the chart such that selecting one of them and typing a value would automatically cause the value to be passed to `growColumn` on that particular rectangle. An even more sophisticated scenario, but still completely possible with a simple programming language facility, would be to configure the program to respond usefully to the user attempting to paste a comma-separated list of values on top of a group of selected rectangles. Each value in the list could be sent to the `growColumn` function for each rectangle, causing the entire set of rectangles to be accurately sized at once.

2.1.3 Learning Challenges

There are a number of formidable barriers that stand between designers like the informants in the CU Publications field study and the above vision of end-user programming. First, one must be concerned that programming is such a different activity from designing that graphic artists would be reluctant to even begin using a computer language, especially non-visual text-based ones, such as seen in the SK8Script examples. After all, both informants had rejected Illustrator's spreadsheet interface, it would seem, for being too foreign. Any language that designers are expected to learn must respect their existing skills and build on these rather than present a radically different means of formally expressing their work. A second barrier is time for learning. The CU Publications informants were busy professionals responsible for juggling a number of clients simultaneously. Any time used to learn new skills, regardless of the long term benefits, would be time taken away from these pressing projects.

Even if designers have the time and interest to learn programming, an additional challenge is developing appropriate educational resources. For example, the field study informants seemed to favor learning experiences that were both informal and highly relevant. The designers generally preferred trying things out for themselves or talking to knowledgeable colleagues to reading the software manuals. When they did turn to these manuals (only two copies of each manual were kept in the department library), it was usually to address specific questions about their current tasks. This task-directed learning strategy mirrors results from related field studies of office workers in general (Mackay, 1991; Rieman, in press). The informants made only limited use of software training classes for similar reasons. Designer A complained that classes taught techniques that were not very relevant to her current projects and Designer B reported that he was having difficulty setting aside time in the evenings or Saturdays to attend them.

The appropriateness of software tutorials for designers is also questionable. Prior to the Illustrator tutorial, neither of the informants in the field study had stepped through any of the tutorials supplied with their design software. When they did try the Illustrator tutorial, they seemed interested in learning about the graphing features, but were also resistant to the tutorial's rigid sequence of steps. Both commented that they would have done some steps differently had they been on their own. A consistent confounding factor, particularly for Designer A, was the need to constantly switch focus from the printed tutorial (placed beside the keyboard) and the evolving design on the screen. Reading the instructions out loud to Designer A seemed to help her considerably.

2.2 Dale Glasgow & Associates

The next case study comes from the book *Information Illustration* by Dale Glasgow (Glasgow, 1994). The author is founder of Dale Glasgow & Associates, a small graphic design firm with well-known clients such as *U.S. News and World Report*, Air & Space Smithsonian, General Motors, Ogilvey & Mather, Bell Atlantic, MCI, IBM, Exxon, Philip Morris, and Champion International. Before forming Glasgow & Associates, Glasgow worked at *USA Today* as an illustrator and graphics editor for the Money section. *Information Illustration* is a how-to guide for graphic artists which describes lessons learned from his more than 10 years of experience doing design on the Macintosh. The 31 case studies in his book describe the steps Glasgow took to create everything from an investment map for a real estate company to the universal health security card proposed by the Clinton Administration. The primary software used to produce these illustrations was Macromedia FreeHand, a drawing package almost identical to Adobe Illustrator, the tool of choice for the designer in the previous case study.

2.2.1 A Custom Pie Chart

One of the more interesting case studies in Glasgow's book, from the perspective of this report, describes the production of an illustration for a retirement magazine. The magazine wanted an attractive diagram showing the recommended dietary division of calories from fat, protein, and carbohydrates. The resulting chart is reproduced in Figure 2-2. In his book, Glasgow describes in detail the steps he took to produce the final design. Most notable is the methods used to create the three pie slices by way of a jury-rigged on-screen protractor. Glasgow first drew a circle and a single vertical line along the radius. Next, he duplicated the line and rotated it 3.6 degrees from the circle's center to create a second slice using the Transform dialog shown in Figure 2-3a. The next figure in the sequence shows how Glasgow then used FreeHand's Duplicate menu command to generate additional slices, each offset by 3.6 degrees from the last. By "keeping the trigger finger on this command," he was able to evenly divide the circle into 100 slices. Glasgow then counted 30 3.6 degree slices on his protractor to identify where one pie wedge would go, then 20 more for the second and third wedges as shown in Figure 2-3c.

According to Glasgow the entire illustration including the draped table cloth look and the figures, which were pasted in from previous drawings, took approximately six hours to produce. Glasgow's trick for creating an on-screen protractor could not have occupied a large portion of this time, but it highlights the sometimes mixed blessings of computer-based design tools. While computers offer powerful features they also sometimes rob designers of very useful traditional tools such as physical protractors. These powerful software features can sometimes be used to construct tools that mimic the traditional ones, as Glasgow demonstrates. However, this is not always the case.

2.2.2 Potential for Programming

What solutions might the end-user programming community offer for Glasgow's pie chart problem? A simple language-based aid would be the ability to call the Duplicate menu command repeatedly automatically, without the need for a "trigger finger," using an expression such as

```
repeat 98 times
   menuSelect the menu named "Duplicate" of the menu named "Object"
end repeat
```

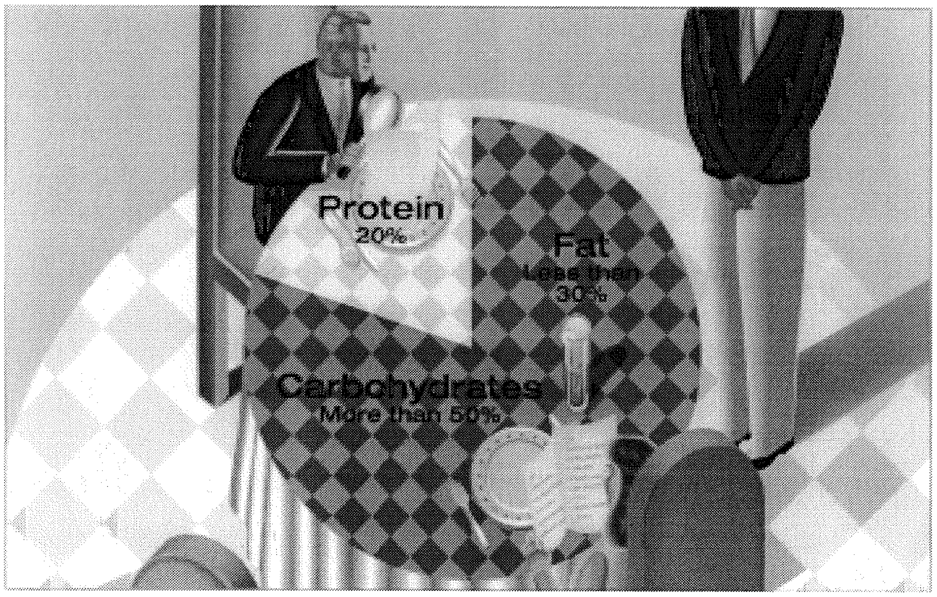


Figure 2-2. The final design illustrating the three main components of a recommended diet.

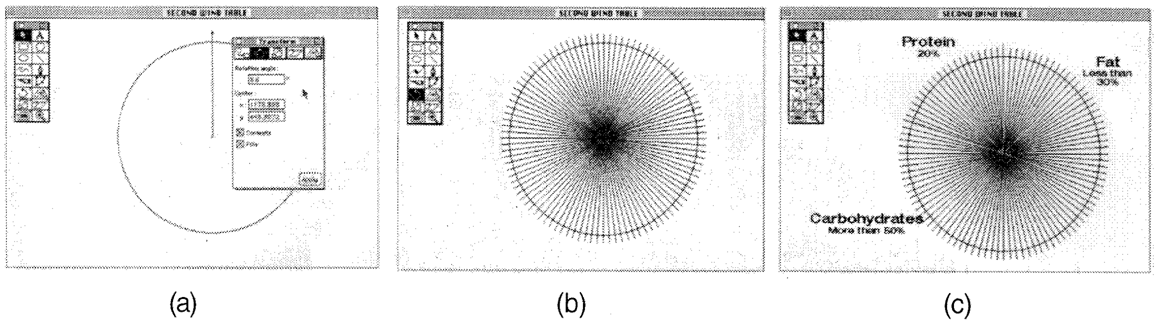


Figure 2-3. Steps in creating an on-screen protractor to produce a pie chart in FreeHand. (a) Starting with a circle and a single vertical line, the line is duplicated and rotated with the Transform dialog. (b) Repeated use of the Duplicate command create the other slices in the protractor. (c) The pie is sliced by manually counting protractor lines.

This type of solution, characterized by "macro languages" is a common facility in productivity packages such as spreadsheets and word processors, although not as prevalent in drawing programs. More sophisticated approaches would obviate the need for an on-screen protractor in the first place. One possibility would involve specifying the interior angle of the wedge and its angle of orientation via mechanisms described in the CU Publications case study. Another approach would be to associate a callback procedure with the drag feedback for arc creation by direct manipulation:

```
on dragFeedback of Arc
   sendToLog my angle
end dragFeedback
```

This procedure could continuously print the current angle of the arc in degrees, radians, or percent as the user dragged out the shape. Although not an automation technique, this approach would empower designers with the ability to create elements more accurately in ways unanticipated by the original tool developer. Such mechanisms would be particularly appropriate for pie charts which typically have a relatively small number of data points.

2.2.3 Learning Challenges

What can be gleaned from *Information Illustration* about Glasgow and his work environment suggests that end-user programming solutions like the ones above could be difficult to introduce into Glasgow's repertoire of design techniques. As with the CU Publications graphic artists, Glasgow seems aware of the existence of spreadsheet or charting software that could automate some of his processes or make them more accurate, but has nonetheless avoided this route. It would seem logical for Glasgow to advocate such tools to produce, if not anything else, a starting point for a custom chart design. After all, in the chapter "Setting up an Electronic Studio" his list of useful software in addition to the "core illustration software" includes spreadsheets and in the same section he writes:

> *...you don't want to rein yourself in, constrain yourself from doing images in certain ways because you don't have the appropriate type of software—and because doing the image with the software you do have would take so long you'd get burned out. (Glasgow, 1994, p. 24)*

And yet, none of the eleven numerically oriented case studies in the book mentions the use of a spreadsheet or charting program.

Assuming that Glasgow could be convinced of the value of end-user programming techniques, his home office work environment presents a unique challenge to one of the most prevalent means of learning about software: seeking help from others. A number of studies about customization and use of software point to help from colleagues as critical for skill advancement (Gantt & Nardi, 1992; Mackay, 1991; Rieman, in press). It should be noted that the communities that were studied consistently included some very advanced users or "gurus" (Gantt & Nardi, 1992). In contrast, Glasgow works primarily by himself in a home office with no nearby programming experts. Opportunities for independent designers like Glasgow to learn end-user programming must take into account the limited social interactions inherent in small office situations.

2.3 The Daily Camera Newspaper

The third case study concerns a custom graph that appears monthly in Boulder's local newspaper, *The Daily Camera*. This graph is found in the *Camera's* "weather wrap-up" section which uses text and pictures to describe the variations in temperature, precipitation, and other meteorological statistics over the previous month. These statistics are compiled by an amateur meteorologist in Boulder who types them on his home computer, prints them out and hand carries them to the *Daily Camera* office. It is then the responsibility of graphic artists at the *Camera* to lay out the information and translate some of the numbers into attractive graphs.

2.3.1 A Floating Bar Chart

One of the more intriguing set of statistics, the high and low temperatures for each day, is depicted with a "floating" bar chart as shown in Figure 2-4. The bottom of each bar denotes the low for each day and the top, the high, with the exact values appearing on horizontal axes above and below the bars. Not only does the graph allow for useful comparisons of highs and lows across days, but the range of temperature is nicely captured in the length of the bar, which, appropriately, also looks a bit like a thermometer. Unlike the charts in the previous case studies which were refined or elaborated versions of standard chart types, the floating bar chart is not supported by most charting packages or spreadsheets (although at this writing DeltaPoint has recently released version 4.0 of DeltaGraph Pro which can do this).

What makes this floating bar chart particularly interesting is that 28 to 31 new highs and lows must be entered into the graph each month, providing an incentive for designers to develop a process which is as efficient as possible. Indeed, the graphic artist currently assigned to this job claims that he can generate a new temperature chart in 15 to 30 minutes. But once again, no automatic tools are used to size or position the chart elements or enter the text values for the horizontal axes. Rather, the designer relies on the general-purpose drawing tool Macromedia FreeHand, the same package used by Glasgow.

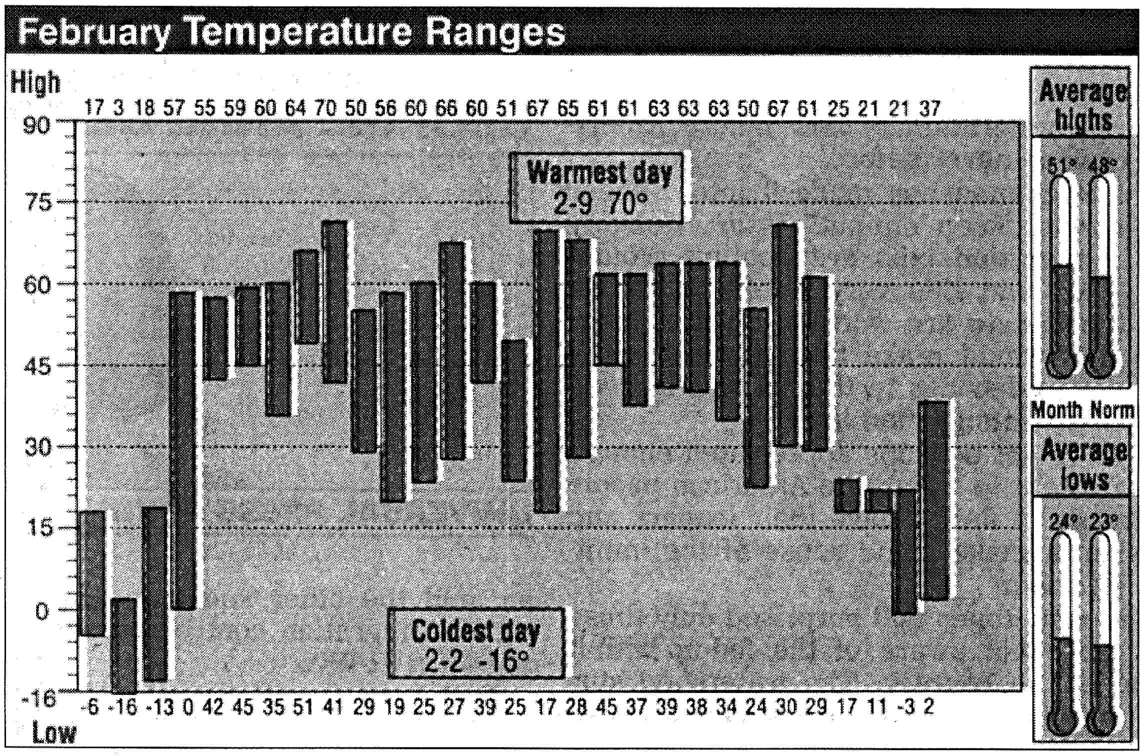


Figure 2-4. The floating bar chart used by the Daily Camera to depict the high and low temperatures for each day in a month.

The key to the design strategy adopted by the Daily Camera graphic artist is to modify the temperature charts from the same month the year before. This approach guarantees that the chart already has the correct number of data points and it also makes it likely that temperatures will be in a similar range, thus limiting the degree of adjustment that needs to be made to the bars. In a phone interview, the designer admitted that occasionally the file for a month from the previous year is missing, forcing him to adapt a different month by adding or removing bars. Changing the exact temperature text in the upper and lower horizontal axes is a matter of placing the text tool on the text boxes and typing the new values. Adjusting the bottoms of each bar involves two steps. First, the designer selects both bottom corner anchor points by clicking on one point then shift clicking on the other. Then he drags the bottom edge up or down and releases the mouse when it seems to be in approximately the right location. To help guide his dragging, the designer looks at the left axis, the dotted horizontal grid lines, and any bars he had recently adjusted for the same month to represent similar temperatures. A similar technique is used to adjust the position of the tops of each bar.

2.3.2 Potential for Programming

The ideal programming solution would query the graphic artist for the month and the new highs and lows and then automatically generate a new chart based on the standard design, completing the task in only the time it takes to enter the new values. But perhaps a more realistic goal for an end-user programmer would be an interplay between direct manipulation and language-based interaction. The user could be responsible for opening the appropriate file for the target month in the previous year, just like is done now. Programming language expressions like those described in the first case study could then be used to adjust the position and size of a bar to correspond to its new high and low temperatures. For example, if the first day of the month had a low of -6° F and a high of 17° F, a user could execute the following expressions (assuming a 1:1 scale):

```
set the bottom of Rectangle 1 in DrawWindow to -6
resizeTop of Rectangle 1 in DrawWindow to 17
```

where resizeTop stretches or shrinks the bar so that its top is at the given position. The real power could come through mapping expressions that could adjust each bar for its respective temperature range:

```
set the bottom of every Rectangle in DrawWindow to ¬
   every item in {-6, -16, -13, 0, 42, 45, 35, 51, 41, ¬
      29, 19, 25, 27, 39, 25, 17, 28, 45, 37, 39, 38, ¬
      34, 24, 30, 29, 17, 11, -3, 2}
resizeTop of every Rectangle in DrawWindow to ¬
   every item in {17, 3, 18, 57, 55, 59, 60, 64, 70, ¬
      50, 56, 60, 66, 60, 51, 67, 65, 61, 61, 63, 63, ¬
      63, 50, 67, 61, 25, 21, 21, 37}
```

This is one of the more straightforward solutions available in languages such as SK8Script that support list mapping, but one can imagine a variety of other solutions involving iteration or recursion.

In order to update the temperature chart on a monthly basis, the user could save the above text in an on-line scrapbook and paste the expressions back into a programmable drawing tool's interpreter each month, replacing the values in the brackets with the new highs and lows for the current month. A somewhat more advanced user could abstract the mapping functions into a named procedure:

```
on adjustBars of highList, lowList
   set the bottom of every Rectangle in DrawWindow to ¬
      every item in lowList
   resizeTop of every Rectangle in DrawWindow to ¬
      every item in highList
end adjustBars
```

which could be loaded and executed each month to produce a new chart. Similar techniques could be used to change the text boxes in the upper and lower horizontal axes.

2.3.3 Learning Challenges

The hurdles that the *Daily Camera* designer would face before he could begin integrating programming in his design activity are in many ways similar to those in the previous two case studies. With his only programming experience being a class in BASIC "a long time ago," the designer would have to be reminded about what programming was and what uses it had. At a minimum, in order to use programming to make his design more exact, the graphic artist would have to learn about expressions for setting shape attributes. To automate his tasks, he would also have to learn about specifying lists of values and using mapping or iteration expressions. Finally, to reuse expressions on a monthly basis he would have to learn strategies for saving and reloading his code, possibly abstracted as a procedure. To motivate this learning, the *Daily Camera* designer would have to be convinced that programming had a larger role in his work than just the temperature chart. In the phone interview he pointed out that the graph was a small part of his work and expressed concern that a programming language would not be relevant to the rest of his projects.

2.4 Collaboratory for Research on Electronic Work

The previous case studies have focused on custom charts and graphs created by professional graphic designers. This final example looks at the custom information graphics needs of a research laboratory where the designers themselves are owners of the data. The Collaboratory for Research on Electronic Work at the University of Michigan (CREW) focuses on the design of new organizations and the technologies that make them possible. Unlike the professional graphic artists in the other case studies, researchers at CREW have considerable programming experience, although none use programming regularly.

A particular area of research at CREW is how time is spent in meetings and the flow of discussion from one topic to the next. CREW researchers have conducted comparison studies of teleconferences with various configurations of communications media, compared the way time is spent in meetings in different countries, and looked at the effects of groupware on the events of a meeting. Spreadsheets are used heavily to record the events in meetings and to compile statistics. But the primary vehicle for quickly communicating activity flow in a meeting, the bubble chart, is created using a hand-held calculator and the general-purpose drawing tool CricketDraw.

2.4.1 A Custom Bubble Chart

Researchers in CREW have developed a custom bubble chart to represent activity during a meeting as shown in Figure 2-5. A bubble chart is a type of diagram supported by only the more sophisticated charting packages. It uses a collection of circles or "bubbles" to depict data where the diameter of each circle is proportional to the value of its corresponding data item. CREW uses these bubbles to represent categories of activity during a meeting such as "Plan and Write," the area of which represents the total time the group spent in that activity. But the CREW version of the bubble chart adds several other data dimensions. The area of each bubble can be divided into sections, the white portion representing the direct introduction of an idea, the black wedge denoting the time spent clarifying the topic. Directed edges between bubbles represents the transitions between activities, the weight of each line proportional to the frequency a transition is made.

Interviews with one of the CREW researchers uncovered a cumbersome but effective process for creating an aesthetically pleasing bubble chart with CricketDraw in approximately two to three hours. First, an "anchor" bubble such as the one for "Alternatives" in Figure 2-5 is drawn using CricketDraw's oval tool. With the ratio of the area of the resulting circle to the time spent in that activity as a reference, the desired

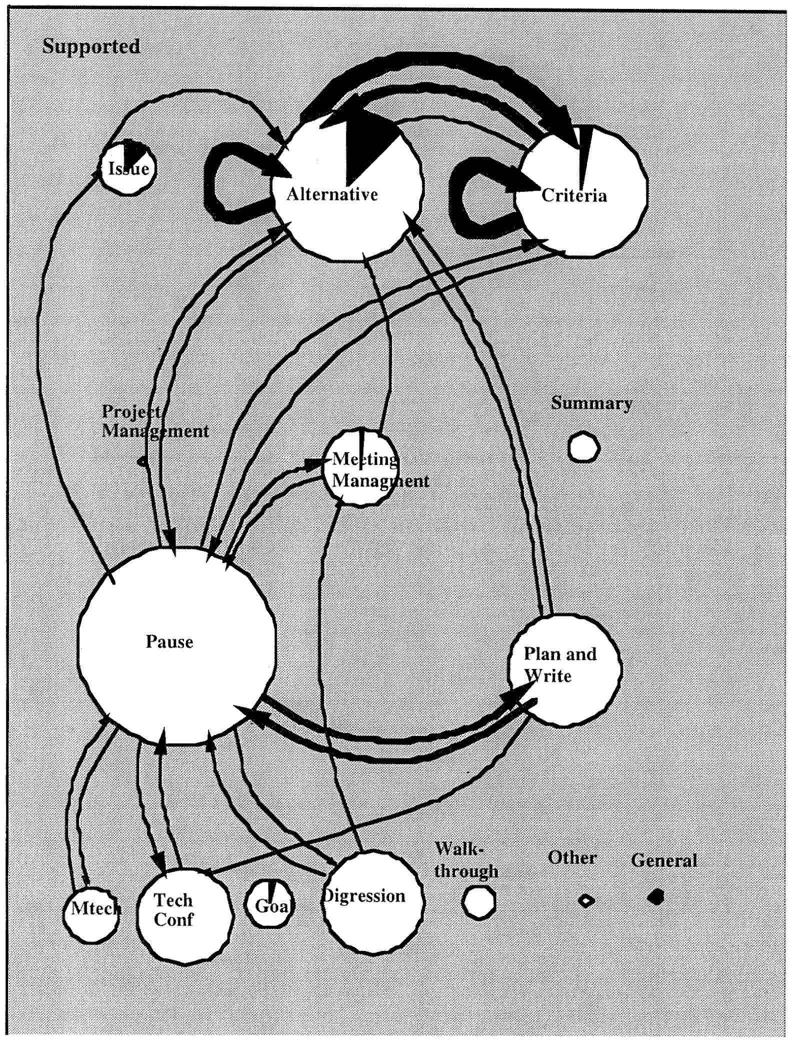


Figure 2-5. A sample bubble chart developed by CREW to depict how time is spent during meetings. This particular chart depicts the timing and flow of activities for a group using a shared editor, hence the title "Supported." The original chart can be found in (Olson, Olson, Storrosten & Carter, 1992).

diameters of the remaining bubbles is computed from the activity data. These bubbles are then drawn to their approximate sizes by using CricketDraw's on-screen ruler as a guide while dragging the oval tool. Next, the bubbles are repositioned so that they are approximately evenly spaced. A black wedge representing topic clarification is then added to the appropriate bubbles. This wedge is drawn by direct manipulation using CricketDraw's triangle tool, for lack of an arc tool. Smaller triangles are used to fill in the area towards the edge of the circle that were missed by the main triangle. The final step involves drawing the curved arrows between bubbles to denote transitions between activities. CricketDraw's splines tool is used to draw the arrows at four to five different pen widths, depending on the frequency magnitude. To avoid cluttering the diagram, only transitions over a certain threshold frequency are sketched.

2.4.2 Potential for Programming

Clearly, end-user programming could play an important role in simplifying the above bubble chart creation process. Programming could automate many of the tedious or inaccurate steps enabling CREW researchers to more quickly and accurately compare meetings. As with the columns in the first case study and the pie slices in the second, a simple but useful programming technique would be the ability to exactly specify some of the dimensions of both the bubbles and edges. For example, if a total of 20 minutes was spent on an activity, one could use the following expressions to adjust the size of a bubble appropriately (assuming a 1:1 scale):

```
set the height of Oval 1 in DrawWindow to sqrt (20 / Pi) * 2
set the width of Oval 1 in DrawWindow to sqrt (20 / Pi) * 2
```

Note that the expression to the right of the token '`to`' automatically computes the diameter from the area, obviating the hand-held calculator. Even more useful would be a procedure that could size any bubble for its corresponding time:

```
on adjustBubble n, a
   set the height of Oval 1 in DrawWindow to sqrt (a / Pi) * 2
   set the width of Oval 1 in DrawWindow to sqrt (a / Pi) * 2
end adjustBubble
```

A constraint-based programming language would be useful to help layout the bubbles after they are sized using an expression such as:

```
given Bubble1 (an Oval), Bubble2 (an Oval)
   declare distanceBetween (Bubble1, Bubble2) - ¬
      (Bubble1's height)/2  - (Bubble2's height)/2 > 30
```

where distanceBetween computes the Euclidean distance between the centers of its two arguments. This expression declaratively specifies that the gap between bubbles must be greater than 30 pixels. The designer could then be responsible for tweaking the arrangement in order to group related activities or to minimize edge crossings. Edges could automatically be placed between activity bubbles using programming:

```
on drawEdge of Bubble1, Bubble2
   make LineSegment with tail Bubble2's center with head Bubble1's
center
end drawEdge
```

A more sophisticated procedure could compute the splines points for curved edges which start and stop short of the bubble's centers as in Figure 2-5.

2.4.3 Learning Challenges

As already experienced programmers, CREW researchers face a different set of challenges in acquiring the particular end-user programming language or languages supported by their design tools. They may need less convincing about the benefits of programming techniques, but they must still be informed of the basic paradigm of the language such as whether it is imperative or declarative and they must still learn the language's specific vocabulary and syntax in relation to their existing programming knowledge. Perhaps most important, CREW researchers would likely need to be trained in the highly interactive style of programming described in the programming solutions in this chapter which are in fact typical for programmable applications. Users with formal training in programming may have to "unlearn" software engineering techniques that encourage planning, modularization, and progressive refinement. In their place users may need to learn more opportunistic strategies that seem successful for programmable applications, such as the intertwining of direct manipulation and language-based interaction.

2.5 Summary

The above case studies touch on a number of important and, in some cases, surprising themes relating to the design of information graphics and the practicalities of introducing programming into such design activity.

- Designers are currently creating custom information graphics with relatively primitive software tools which force them to position and size graphics "by eye." This may come as a surprise to viewers who have assumed graphs like the one in Figure 2-1 were plotted extremely accurately by automatic charting tools.

- Designers often encounter serious limitations with their drawing software such as the lack of a protractor tool in FreeHand, but are able to develop clever "work-arounds."

- A simple end-user programming language can provide the necessary abstractions to accomplish many of their tasks more accurately and can help automate repetitive jobs such as producing monthly temperature charts.

- Designers are busy people with limited time for learning.

- A key challenge in orienting designers to end-user programming languages is situating the instruction in terms of tasks which are of interest to designers.

- Programming language instruction cannot rely on the existence of social networks or the necessary organizational infrastructure that would enable users to consult with a guru or attend training classes.

3. Self-Disclosure

The case studies in the previous chapter illustrate the limitations of direct manipulation tools for information graphics and highlight the potential for programming to free designers from those constraints. The solutions offered for each case study suggest how designers might weave programming expressions into their regular interaction with drawing tools, enabling them to accomplish tasks faster and more accurately, and to take on whole new kinds of problems they had never considered previously. However, for many users the transition from a direct-manipulation style of interaction to a more formal, language-based one can be a daunting challenge. Consider how designers in some of the case studies were reluctant to even use spreadsheet programs—software thought by many to be one of the most approachable end-user programming environments (Lewis & Olson, 1987). It is also not clear how readily designers with programming experience, such as those in the CREW case study, would make the transition from the formal use of standard general-purpose languages such as FORTRAN to the more ad hoc application (as seen in the programming solutions offered in the previous chapter) of the domain-enriched language embedded in a programmable application. The issue of transition for both beginning and experienced programmers represents the central "learnability problem" of programmable applications.

This research seeks to address the language learnability problem inherent in programmable design tools by making these systems "self-disclosing." Self-disclosing programmable tools present short, relevant programming language examples as they are being used. By situating these incremental language learning opportunities within the context of authentic activity, this report hypothesizes that users can acquire end-user programming skills with minimal distraction from everyday activity. This chapter defines the notion of self-disclosing computer systems in its broadest sense and develops a framework for describing various forms of self-disclosure when the technique is applied for educational purposes. This framework is used to compare and contrast existing self-disclosing systems, and is also employed to characterize the prototype self-disclosing design tool, Chart 'n' Art, after the system is described in Chapters 4 and 5.

3.1 Self-Disclosure Defined

The term "self-disclosure" is most often found in psychological and sociological literature where it refers to the act of revealing private information about oneself to a confidant or to the general public (c.f. Derlega & Berg, 1987). The term self-disclosure applied to computers, on the other hand, is a much rarer occurrence, perhaps because authors are reluctant to use the word "self" for fear of anthropomorphizing a machine. However, this expression can be very appropriate. An early example comes from Bolt in his book, The Human Interface:

> *Imagine: a computer-based body of information that unfolds itself to you automatically as a function of the curiosity and interest you show in it. This process of* self-disclosure *goes in both directions: The information base discloses itself to you as you disclose yourself to it. (Bolt, 1984, pp. 86-96)* (original emphasis)

The author then describes eye tracking techniques that could be used to determine what on the screen was particularly interesting to users. Another instance of the term "self-disclosure" is found in (Venolia, 1994). Here the authors use the term when describing a helpful feedback mechanism they have built into a pen-based interface: When users hold down the pen at any point in the middle of a gesture the system reveals a "pie menu" of possible next strokes and their meanings.

A subtle difference between these two uses of the term and the psychological definition of self-disclosure is that the party doing the disclosing, in this case a computer, is not really doing so on its own volition. A curious glance or deliberate pause by the user is what actually triggers the feedback in these systems. Closer to its psychological roots would be a definition for self-disclosure in computers that encompasses the notion of *information volunteering* (Fischer & Stevens, 1987); that is, presenting information that was not requested and possibly not even anticipated by the user.

Bolt's use of the expression "self-disclosure" also differs from the psychological definition in another way. Bolt's imaginary system discloses information external to the computer, such as—to use one of his examples—a browsable image of an historic house. In contrast, self-disclosure in psychology is about revealing very personal *internal* information. The nature of a computer's internal information depends on one's perspective on what private data a computer could be said to own. This report will consider a computer's internal information to be the hidden or non-obvious information about its status or functionality. An example of this kind of internal information is the pen gesture functions revealed by Venolia and Neiberg's self-disclosing interface.

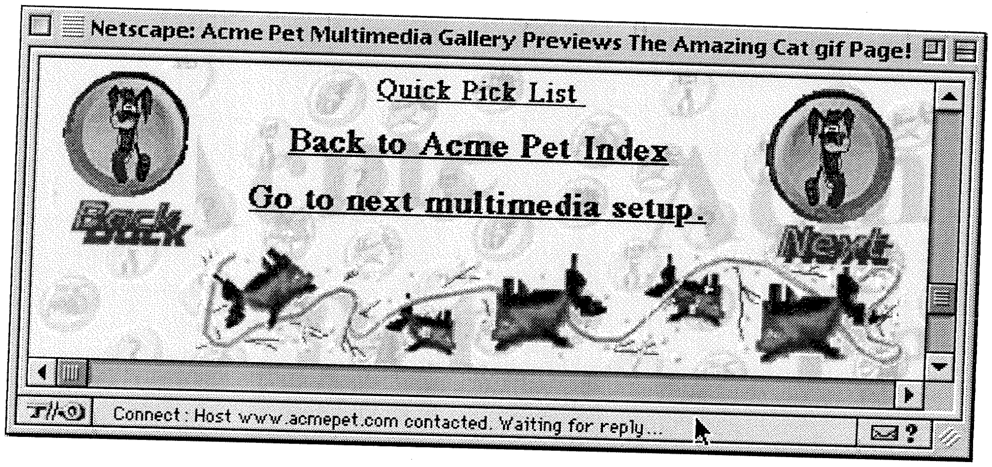
Combining the notions of information volunteering and internal information leads us to a general definition of self-disclosure in the context of computers that captures much of the essence of the term's psychological meaning:

> *Self-disclosure is the act of the computer volunteering hidden or non-obvious information about its status or functionality.*[3]

3.2 Examples of Self-Disclosing Systems

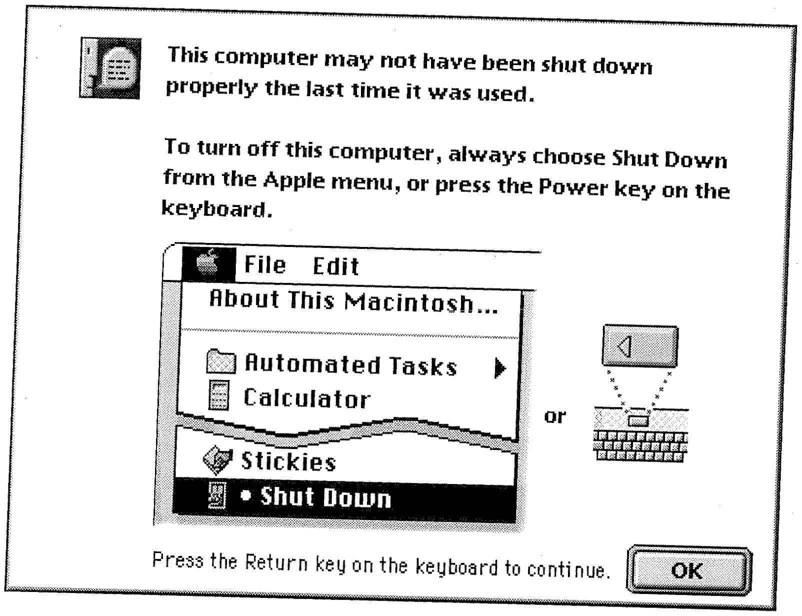
Self-disclosing systems as defined above are not uncommon, although they are sometimes described using other terms such as "transparent," "glass box," (Boulay, O'Shea & Monk, 1981) or "open" systems. Figure 3-1 depicts some very simple feedback mechanisms that fit the definition of self-disclosing systems. Figure 3-1a is a snapshot of the Netscape Navigator web browser after the user has just clicked on the link "Back to Acme Pet Index." While Navigator is attempting to retrieve the data for this link, it volunteers information about the status of the necessary network connection at the bottom of the window.

[3] Note that this is a working definition that suffers from a number of ambiguities. For instance, whether or not a computer is "volunteering" information may be subject to individual interpretation. Likewise, information that might be obvious to one user may not be so obvious to another. Thus, a system that reveals such information might seem self-disclosing to only certain individuals.


Netscape: Acme Pet Multimedia Gallery Previews The Amazing Cat gif Page!
Quick Pick List
Back to Acme Pet Index
Go to next multimedia setup.
Back
Next

Connect: Host www.acmepet.com contacted. Waiting for reply...

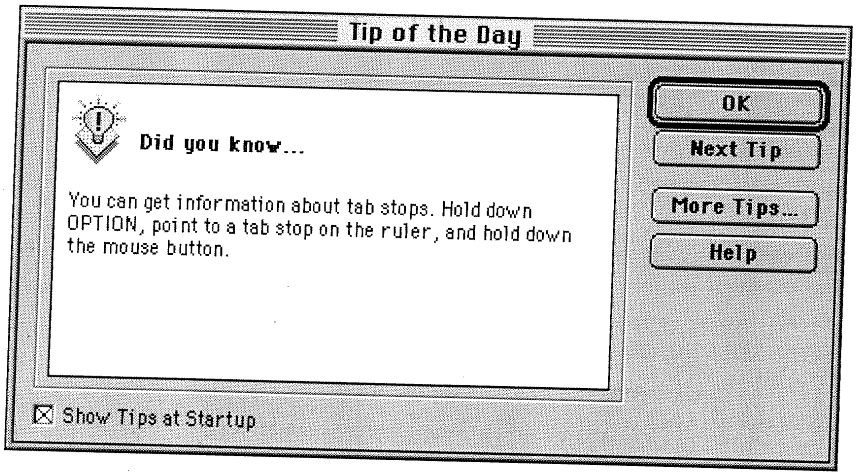
(a)


This computer may not have been shut down properly the last time it was used.

To turn off this computer, always choose Shut Down from the Apple menu, or press the Power key on the keyboard.

Press the Return key on the keyboard to continue.

(b)



Did you know...

You can get information about tab stops. Hold down OPTION, point to a tab stop on the ruler, and hold down the mouse button.

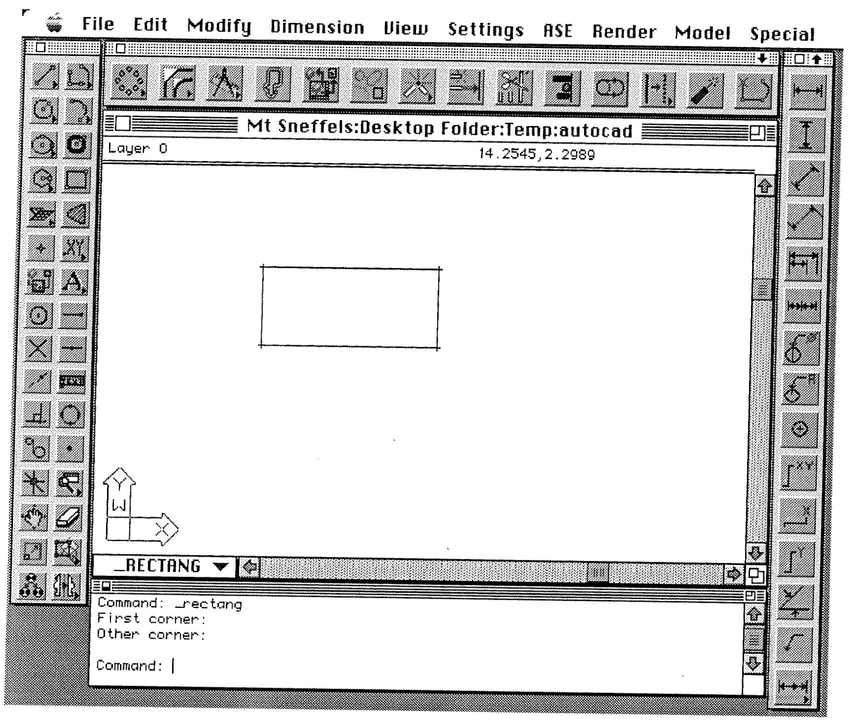
☒ Show Tips at Startup

(c)

Figure 3-1. Examples of self-disclosing systems: (a) Netscape Navigator discloses internal network status in a line of text at the bottom of its window. (b) The Mac OS discloses what some might consider non-obvious functions for shutting down the computer. This dialog box appears when starting up the computer after "improperly" turning off the machine by pulling its plug. (c) Microsoft Word's Tip of the Day discloses how to inspect tab stops.

Figure 3-1b is the dialog box that can appear when users start up a computer running the Macintosh Operating System (Mac™OS) after the machine had literally been unplugged. Here, the computer is volunteering information, in this case about what some might consider the non-obvious ways to properly shut down a Macintosh. Another example of self-disclosure that is familiar to many personal computer users is the Tip of the Day dialog shown in Figure 3-1c, which is available in some Microsoft productivity tools such as Word (Microsoft Corp., 1996c). The Tip of the Day messages consist of random hints on how to use a tool's advanced features, especially those not advertised in menu items or on-screen buttons. Users can configure their Microsoft tool to display the Tip of the Day window each time the software is launched.

A number of programmable tools also feature disclosure mechanisms. An example of a self-disclosing programmable tool is the computer aided design system, AutoCAD, shown in Figure 3-2. The interface for this tool has evolved over the years from a simple command line to a direct manipulation-style graphical user interface. 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. The screen snapshot in the figure was taken immediately after the user drew a rectangle on the screen using the mouse. Notice how the system has volunteered the normally hidden command "_rectangle" by printing it in the Command window.


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 3-2. AutoCAD's self-disclosing interface. Drawing a rectangle with the mouse reveals the command _rectangle in the bottom Command window.

Note that there are a number of systems related to the above examples that do not qualify as self-disclosing under the given definition. For instance, on-line help systems that require users to explicitly request assistance via a keystroke or menu command, are not self-disclosing, since information is not being volunteered by the system. Most critiquing systems, although they do typically volunteer information, would also not qualify as they present feedback about an application domain such as kitchen layout (Fischer & Mørch, 1988) or the use of color in drawings (Nakakoji, Reeves, Aoki, Suzuki & Mizushima, 1995) as opposed to a tool domain. Even though the data structures for such critiquing knowledge are internal to the system, the entities to which they refer (e.g. kitchen appliances, human color perception, etc.) are *external* to the computer. Such critiquing systems could be considered "disclosing," but not "self-disclosing."

3.3 Learning Programming from Disclosures

Disclosures are by definition informational in that they communicate data about a system's status or functionality for all users to see. The more interesting issue, however, is whether disclosures can be not just informational but *educational*. For example, after Word users see the Tip of the Day in Figure 3-1c can they remember a tip (especially after seeing the same tip more than once) and eventually employ it in their editing? Of specific interest to this research is the potential for a user to learn programming expressions through disclosures. The following anecdote illustrates how self-disclosing programmable applications have provided (in at least one instance) programming language learning experiences to their users:

> *An experienced architect and graphic designer 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 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. (DiGiano & Eisenberg, 1995)*

The educational feedback provided by AutoCAD in this anecdote illustrates a particular role for programming language disclosures called *echoing*. Echoing is when a system reveals programming expressions that could replace mouse actions. This report identifies three other educational roles for programming language disclosures: *advertising*, *identifying*, and *monitoring*. All four techniques are summarized in Table 3-1. Advertising is when a system presents a set of possible expressions that could be used to operate on various objects on the screen. Identifying is when a system discloses expressions for referencing an object. Finally, monitoring is when a system dynamically shows the connections between graphical specifications (typically through on-screen forms) and programming language declarations or specifications. Chapter 4 illustrates the four disclosure roles in the context of Chart 'n' Art.

Table 3-1. Four disclosing techniques for providing programming language learning opportunities to users.

Disclosing Technique	Description	Example
Advertising	presenting possible expressions that could be used to operate on various objects on the screen	The user selects an oval in a drawing and the system discloses commands for operating on ovals.
Echoing	revealing language expressions that could replace mouse actions	The user drags out the shape of rectangle and the system discloses a rectangle creation command.
Identifying	revealing expressions for referencing an object	The user selects a rectangle and the system discloses a unique name for that shape.
Monitoring	dynamically showing the connections between graphical specifications and programming language declarations or specifications.	As the user interacts with a dialog box for aligning shapes, the system updates a linked textual expression for aligning shapes.

Even with AutoCAD's limited self-disclosure mechanisms, the anecdote demonstrates how a user can acquire at least rudimentary knowledge about a programmable application's programming vocabulary, syntax and semantics. The story also demonstrates the central characteristic of the educational experience offered by self-disclosing systems in general, that learning is incidental to the use of the system. Users of self-disclosing tools can be exposed to potentially educational feedback during their regular activity, so that learning is not necessarily a distinct enterprise. For example, architects like the one in the anecdote can learn while designing; mathematicians can learn while solving; users can learn while doing.

Learning programming "while doing" (Anzai & Simon, 1979) or "on demand" (Fischer, 1991) is in stark contrast to the more traditional programming language opportunities such as tutorials, training classes, manuals, or personal consultation with an expert or "guru." These educational resources along with others to be discussed later in this chapter are presented in Table 3-2. For busy designers such as those in the case studies, these traditional approaches are undesirable because of the time they take away from designing. Some work environments such as the CU Publications Department may further decrease designers' willingness to engage in traditional training experiences by encouraging employees to pursue self-improvement only on their own time. An additional drawback, as one of the informants at CU Publications complained, is that traditional learning opportunities are typically decontextualized from the projects in which designers are actually involved. Lack of relevance is also a common problem with many computer-based learning opportunities such as on-line help (Houghton, 1984). In contrast, educational self-disclosing tools have the potential to react to designers' current constructions and provide useful programming language learning opportunities that are highly relevant to the "tasks at hand" (Fischer & Nakakoji, 1991).

Table 3-2. A comparison of learning opportunities by relevance and timing. Contextualized opportunities are related to the tasks at hand and synchronized opportunities are those which can take place while performing these tasks.

	Relevance	
Timing	*Contextualized*	*Decontextualized*
Synchronous	self-disclosure critics	on-line help pedagogical screen savers
Asynchronous	consultation	tutorials training classes manuals

3.4 Dimensions of Educational Disclosures

Self-disclosing tools cannot guarantee learning; they can only offer learning opportunities. This research argues that the quality of the learning experience depends critically on the design of disclosures and on the kind of time and attention users can devote to them.[4] One of the goals of this research was to develop a model of the disclosure design space in order to classify existing self-disclosing systems and to gain insight into the how designers of future systems could influence educational impact. This model, summarized in Table 3-3, is comprised of the following six dimensions: adaptivity, explorability, intrusiveness, relevance, scope, and structure.

Table 3-3. Six characteristic dimensions of pedagogical disclosures.

Disclosure Dimension	Description
Adaptivity	the capacity of a self-disclosing system to change the content and form of disclosures to suit the needs of each user individually.
Explorability	the degree to which a self-disclosing system supports active user inquiry into the meaning and utility of disclosures.
Intrusiveness	a measure of how distracting disclosures are for users.
Relevance	the degree to which disclosures are contextualized to user activity.
Scope	the extent of hidden information that is disclosed to users.
Structure	a measure of the strictness of sequence in the presentation of disclosures to users.

The choice of dimensions of educational disclosures was largely influenced by two bodies of research: learner-centered educational computing and human language acquisition. Learner-centered education is a recent pedagogical movement that promotes the same notion of learning-while-doing that is at the heart of the self-disclosure

[4] A related issue is whether or not users need to necessarily remember what they see in a disclosure if they can be assured that the disclosure will appear again when needed. This notion of "using on demand" is briefly explored in the context of Chart 'n' Art in Section 4.4.

approach. This movement emphasizes that "people learn best when engrossed in the topic, motivated to seek out new knowledge and skills because they need them in order to solve the problem at hand." (Norman & Spohrer, 1996) This educational philosophy is also sometimes described using the terms "constructionism," "problem-based learning," or "learning on demand."

Research in human language acquisition has also influenced the development of self-disclosure dimensions. The reason is that learning about a system by observing disclosures can be viewed in many ways as analogous to learning to speak by listening to the ambient language in one's environment. Using language acquisition terminology, one can think of the AutoCAD designer in the above anecdote as being "immersed" in a language rich environment in which the "input language" was AutoCAD commands. Even though child language acquisition typically emphasizes spoken language learning whereas programming languages are traditionally formal written notation, the field provides a rich body of empirical data that has relevance to language learning in general. A related influence on this research comes from the field of second language acquisition, which focuses on older learners with existing linguistic competence. These learners share some of the same motivational challenges faced by those learning programming, since both groups are acquiring new communication strategies that have confusing similarities with and differences from their existing expertise. Of particular relevance to the self-disclosure approach to learning programming is work on incidental second language acquisition from picture books by Richards and Gibson (Richards & Gibson, 1945) and, more recently, on captioned television by Neuman and Koskinen (Neuman & Koskinen, 1992), both of which are detailed below.

Let us now consider each of the dimensions of educational disclosures in turn.

3.4.1 Adaptivity

Adaptivity is the capacity of a self-disclosing system to change the content and form of disclosures to suit the needs of each user individually (Totterdell, Norman & Browne, 1987). A self-disclosing system which is highly adaptable monitors a user's attention to disclosures (perhaps by timing pauses in activity coinciding with the appearance of disclosures or by measuring the proximity of the mouse pointer to disclosures) as well as any history of the user employing disclosed functionality. This data helps form a model of the user's current interests and abilities, and, when combined with a model of the domain of information being disclosed, determines the most appropriate set of disclosures to follow. A self-disclosing system which is not adaptable makes no such attempt to alter its disclosures based on user interaction.

A programmable tool that is highly adaptable might adjust future disclosures depending on how often certain types of expressions (e.g. assignment statements, control structures) had been disclosed and the frequency with which a particular user employed them. For example, if AutoCAD were adaptable, it might initially disclose the simplest form of the `_rectangle` function as shown in Figure 3-2, but after a user called this function successfully a number of times, the system might then show a more sophisticated form of the command which included optional parameters.

Adaptivity is one of the main ideas behind intelligent tutoring systems (ITS), the umbrella term for a variety of today's educational computing efforts. An ITS, in contrast to earlier computer aided instruction (CAI) systems, can generate and adapt teaching materials as it is being used. An example of an adaptive ITS system is Anderson's Lisp tutor (Anderson, Corbett, Koedinger & Pelletier, 1995) (discussed in more detail in Chapter 5) which reacts to an individual student's progress on a programming problem by helping to fill in perceived gaps in the student's cognitive model of the solution. Anderson et al. claim that students using the Lisp tutor can achieve at least the same level of proficiency as conventional instruction in one third the time. Note, however, that the Lisp tutor's adaptivity is just one of a number of key features in the system such as the use of scaffolding and the immediacy of feedback that may have contributed to this rate of success.

Another example of adaptive systems for learning programming is Selker's COgnitive Adaptive Computer Help (COACH) (Selker, 1994). COACH is an open environment for writing Lisp programs that offers advice on syntax and style as users compose expressions, the feedback being determined by a dynamic user model based on perceived experience and proficiency. Selker reports that subjects composed five times as many functions than a control group using the same system without adaptive help, and that the experimental group's code was of higher quality. A less rigorous study of a related adaptive system for aiding Lisp programming, the Lisp-Critic, reported similarly positive results (Fischer, 1987). Both systems are described in more detail in Chapter 5.

Although the evidence from educational computing indicates that adaptivity is important for learning, linguistic investigations suggest that it might not be necessary, at least for child language acquisition. A thorough summary of this research by Snow (Snow, 1986) reports that in North American and Northern European families people speaking to children "fine-tune" their speech by adjusting the syntactic and semantic complexity to match a child's language competence. But Snow also points out that other studies question the importance of such graded input, since children have been found to automatically filter out speech which is beyond their current comprehension levels to avoid becoming confused or misled by complex utterances. Furthermore, caregivers in other societies such as the Kipsigis of Kenya are much less willing to adapt adult speech for their children and yet these children acquire language just as rapidly as in western cultures. Snow speculates that fine-tuning may be more effect than cause, that is caregivers are simply adjusting their speech in order to communicate with a child as it gains linguistic proficiency independent of graded input.

In the ideal, then, experience in educational computing suggests that programming language disclosures should be adaptive to individual users. On the other hand, if learning programming via disclosures is analogous to first language acquisition, this adaptivity may not be completely necessary.

3.4.2 Explorability

Explorability is the degree to which a self-disclosing system supports active user inquiry into the meaning and utility of disclosures. Unlike the previous dimension, explorability is not actually a measure of disclosures themselves, but of the environment that is doing the disclosing. By supporting exploration, self-disclosing systems can broaden the range of educational opportunities they offer beyond the simple incidental learning provided by disclosures alone. A self-disclosing system that offers a high degree of exploration enables users to review previous disclosures, study explanations for disclosures or related information, and, in the case of disclosed functionality, actually try out the revealed commands.

By definition, self-disclosing programmable tools have a certain degree of explorability built-in, since these tools provide a programming language interpreter in which users can try out disclosed expressions. However, to further encourage exploration self-disclosing programmable tools can also offer, among other things, on-screen documentation for each disclosed programming expression, the ability to modify disclosures and execute them directly, and support for undoing executed statements that had undesirable results.

The opportunity to experiment with disclosed programming expressions resonates with progressive educational paradigms such as learning by doing (Anzai & Simon, 1979), learning on demand (Fischer, 1993), constructionism (Papert, 1991), and cognitive apprenticeship (Collins, Brown & Newman, 1989) that emphasize the need for learners to actively employ new knowledge. Evidence is growing that these approaches in the school setting can have positive effects. Resnick's review of educational programs found that, among other things, the successful programs stressed active participation by learners (Resnick, 1989). Support for active experimentation during learning is also found in the professional setting. In their study of professional programmers learning new systems, Berlin and Jeffries identified a common and successful learning strategy which they called "copy and experiment" (Berlin & Jeffries, 1992). They noticed that programmers often duplicated related code and then experimented with it in order to understand how to use the new system. In a related field study, Rieman (Rieman, in press) also found that exploratory learning was a widely accepted strategy; however, an interesting caveat was that users favored it only as long as the exploration was task-oriented.

Language acquisition research provides mixed evidence for the value of explorability in learning programming. Studies of first language learners indicate that the amount of talking a child "needs to do" in order to acquire a language varies considerably: some can speak very little and still become as competent as others who are constantly chattering (Aitchison, 1985). On the other hand, for second language learning explorability seems more important. Studies of second language incidental vocabulary learning report more effective acquisition when learners must retell stories they read which had novel vocabulary (Joe, 1995; Newton, 1995). These studies also observed that students with the greatest improvement used the story vocabulary more generatively, that is in new contexts and in new structures.

What does all this mean for the design of pedagogically effective self-disclosing programmable tools? Generally, it seems valuable for users to be able to become active participants in programming language acquisition, not just passive learners through incidental exposure to a language. Rieman's study implies that as much as possible, users should be able to do this active inquiry into issues directly relevant to their current task.

3.4.3 Intrusiveness

Intrusiveness is a measure of how distracting disclosures are for users. A self-disclosing system with high intrusiveness ensures that users cannot ignore disclosures, for example the shut down warning in Figure 3-1b and the Tip of the Day in Figure 3-1c, which require that users dismiss a dialog box before they can begin working. A system with low intrusiveness presents disclosures subtly, for instance the small status line at the bottom of Navigator's window in Figure 3-1. In the context of self-disclosing programmable design tools, intrusiveness gauges the degree to which programming language feedback diverts users from their design activity.

Obviously, a minimum level of perceptibility is necessary for users to begin learning from programming language disclosures. On the other hand, highly intrusive language disclosures might disrupt design "flow" (Csikszentmihalyi, 1990) to such a degree that the benefits of learning programming techniques are outweighed by the costs. Intrusive but highly valuable learning experiences can lead to a "production paradox" (Carroll & Rosson, 1987) in which learning is inhibited by lack of time and working is inhibited by lack of knowledge. An example of such a quandary is when a system discloses very useful functionality to the user through an elaborate on-screen tutorial that demands the user's complete attention for an extended period of time. The user knows that by following the tutorial he or she could learn important and useful facts about the system, but at the same time is concerned by the disruption it will have on pressing work.

The pedagogical tradeoffs of intrusiveness are well known to designers of critiquing systems for learning. These systems attempt to feed back helpful information about a construction in progress, providing opportunities for users to "learn on demand" (Fischer, 1991). Fischer et al. claim that the principle challenge for critiquing systems is "saying the right thing at the right time" (Fischer, Nakakoji, Ostwald, Stahl & Sumner, 1993). Their work suggests that there is no one proper level of intrusiveness, but that such systems must offer a variety of presentational mechanisms, possibly under user control (Nakakoji, Sumner & Harstad, 1994).

Although intrusiveness in systems for learning while doing is clearly a complex issue, language acquisition provides evidence that a remarkable amount of learning can take place with almost imperceptible feedback from the environment. Studies of children listening to stories (Elley, 1989) and watching captioned television (Neuman & Koskinen, 1992) indicate that students pick up vocabulary incidentally with impressive speed. According to Elley, story telling with teacher explanation resulted in vocabulary gains of 40 percent on a test of target words in a particular story, but even without explanation students gained 15 percent simply from hearing the story. Neuman and Koskinen's study indicated that language minority students can not only acquire second language vocabulary by being exposed to captions, but that they learn more words than if they had read the equivalent text without the video.

Clearly, vocabulary acquisition is only one of the challenges facing language learners. However, numerous formal models of language acquisition exist that are capable in principle of inferring sophisticated grammars from relatively unintrusive language input similar to that provided to the children in the above vocabulary studies (Pinker, 1979). If programming language learning is indeed analogous to first or second language learning, the implication is that even the most unintrusive programming disclosures might prove beneficial.

3.4.4 Relevance

Relevance is the degree to which disclosures are contextualized to user activity. A self-disclosing system with high relevance attempts to time disclosures to coincide with related user activity. A system with low relevance essentially presents status and functionality information randomly to users and makes no effort to connect the feedback to current activity. This is the case with Microsoft's Tip of the Day and an earlier system, DYK (Owen, 1986), which presented random Unix tips upon the user's request. Highly relevant programming language disclosures include expressions that relate to how a user is currently employing a direct manipulation interface (as in AutoCAD) or ones that are known to be particularly useful given the current state of a user's project. Programming language disclosures of low relevance include random expressions intended to be shown during idle moments. Somewhere in between these two extremes along the relevance dimension is the Chart 'n' Art pedagogical screen saver (DiGiano & Eisenberg, 1996) which randomly suggests a variety of commands that could modify the user's current construction in interesting ways.

Research in educational computing gives reason to believe that relevance can be an important factor in the effectiveness of disclosures for learning programming. Often cited is WEST, an intelligent arithmetic expressions tutor for a mathematical game, which attempted to intervene with "relevant and memorable" hints and suggestions (Burton & Brown, 1976). Experiments showed that students exposed to the tutoring exhibited "a considerably greater variety of patterns" than those without tutoring. It is important to note, however, that this level of relevance may only be practical in highly constrained domains such as WEST's mathematical game. More open learning environments may find it much more difficult to anticipate appropriate feedback. Even if the pragmatics of offering highly relevant feedback was not an issue, it is not clear that this is necessarily the best approach. Random disclosures such as in the pedagogical screen saver have the advantage that users can be introduced serendipitously to new and interesting programming language features that might never have been revealed otherwise because of their irrelevance to users' current constructions.

Language acquisition research also offers some insights into the role of relevance in learning from programming language disclosures. For example, Snow's summary of child-directed speech reports that within North American and Northern European families a child learns better when exposed to utterances that express his or her own intentions or relate to topics of the child's choice. On the other hand, the Kipsigis of Kenya do not use such child-centric speech, but Snow argues that their children acquire language just as rapidly because of the prevalence of repeated highly predictable situation-specific adult speech. These results suggest that programming language disclosures might be more learnable if they are either highly relevant to user activity or else repeated in highly predictable ways.

3.4.5 Scope

Scope is the extent of hidden information that is disclosed to users. A self-disclosing system with wide scope exposes users to a large variety of information, in the most extreme case providing coverage over all internal status or functionality information. A self-disclosing system with narrow scope offers only isolated glimpses into what might be going on within. For example Navigator's status line in Figure 3-1 only discloses basic connection information and leaves out lower-level TCP/IP negotiation. In the context of self-disclosing programmable design tools, scope refers to both the variety and complexity of the syntax and semantics of programming language expressions shown to users. For instance, do disclosures illustrate the key concepts (perhaps defined by chapter headings) in programming textbooks such as data structures, control structures, variables, etc.? Do disclosures show multiple-line programs that illustrate program plans (Spohrer, Soloway & Pope, 1985) in addition to simple one-line statements? Do disclosures illustrate more than one way of accomplishing the same task, but using different strategies or syntax?

Obviously, a self-disclosing programmable design tool which could teach an entire language would be very useful. And clearly, a certain minimum coverage of a language is necessary for users to begin employing it in their work. On the other hand, self-disclosure is likely only one of many resources including textbooks and on-line help by which users can learn programming (see Table 3-2). The key issue is whether disclosures provide sufficient examples for designers to begin integrating programming into their repertoire of tools.[5] These examples need not necessarily be that extensive,

[5] A related question is whether designers are motivated by disclosure to take advantage of these alternate resources and learn more about programming.

depending on the design of the end-user language. Powerful yet simple languages such as described in (Eisenberg, 1995) demonstrate that a well-designed end-user language for a particular domain (e.g. painting, simulations of dynamical systems, etc.) can provide a wide range of expressiveness through only a small number of high-level primitives. In addition to careful language design, developers of self-disclosing systems can limit the scope of necessary disclosures by analyzing the patterns of language use in designers already familiar with the system. Such analysis might include a "programming walkthrough" (Bell et al., 1994) of the language interface to determine the "critical mass" of programming knowledge for that particular programmable tool.

Self-disclosing programmable tools can achieve a relatively wide scope by disclosing multiple line examples which illustrate a number of programming concepts at once. This approach is in contrast to the "one-liners" offered by AutoCAD. Note, however, that the disclosure of these longer, richer examples may come at the price of higher intrusiveness and lower comprehensibility. Long examples can easily clutter the screen, potentially covering up valuable context and interrupting design "flow." Even when such intrusion is tolerated by designers, care must be taken not to overwhelm them with a large variety of new material.

For lack of applicable linguistic evidence to the scope dimension, let us now move on to the last of the six dimensions, structure.

3.4.6 Structure

Structure is a measure of the strictness of sequence in the presentation of disclosures to users. A highly structured self-disclosing system is programmed to show specific disclosures or groups of disclosures in a certain order. A self-disclosing system without structure makes no attempt to constrain the order of presentation and must craft disclosures to be understood in isolation.

A highly structured self-disclosing programmable tool presents programming expressions in an order that follows a certain programming curriculum, for example, from basic syntax to control and data flow to sophisticated optimizations. Note that such an environment does not preclude the possibility that the feedback could still be highly relevant to users' current activities. For instance, in AutoCAD the use of a tool could trigger a disclosure thematically related to the current point in the curriculum but also temporally connected to the particular tool currently being used.

The most pedagogically effective level of structure is still a matter of contentious debate among educational computing researchers. In part, this is due to varying definitions on what it means to be "pedagogically effective." Early CAI systems emphasized a rigid curriculum that indeed enabled certain individuals to perform better on standardized tests. However, these systems were criticized for teaching decontextualized knowledge through demeaning drill and practice. ITS was developed to address some of these problems by adding the ability to adapt materials somewhat for the individual learner, but at its heart is still a fixed curriculum based on a model of expert behavior. On the other end of the structure spectrum lies the *open learning environment* popularized by Papert in (Papert, 1993). Open learning environments are foundational to constructionist educational philosophy. Constructionist systems provide little, if any, structure and instead rely on the self-directed learner to find "cognitive hooks" or touchstones for learning. Case studies of constructionist approaches cite anecdotal evidence of limited success, but few, if any, formal evaluations have been conducted (Elsom-Cook, 1990). Increasingly, researchers are seeking a middle ground between the two extremes of ITS and constructionism as is evidenced by new approaches such as learning on demand (discussed previously) and guided discovery tutoring (Elsom-Cook, 1990).

The degree to which structure is important in human language acquisition is also a complicated and controversial topic. As mentioned above, Snow's research suggests that although language input for many children is "graded," it is not clear whether this type of structure is necessary, since children can automatically filter out overly complex speech on their own. One thing is clear, however: Caregivers rarely, if ever, offer children explicitly sequenced grammar or vocabulary lessons. Rather, human language is somehow acquired *implicitly* from exposure to a wide variety of child directed speech.

Although educational computing and language acquisition offer no obvious suggestions for the optimal degree of structure for programming language disclosures, they do teach two important lessons: (1) the appropriate amount of structure is dependent on the desired learning experience, and (2) learning can potentially take place in the absence of structure. If a user has a casual interest in programming, a very loosely structured set of disclosures—not necessarily all noticed by the user—might be adequate for long term acquisition. Considerably more structure might be necessary if the user needs to learn programming rapidly. Note, however, that the structure can only be so rigid and fine-grained before users begin missing important elements in the sequence due to the fact they are still trying to get work done with their tool and cannot always attend to disclosures.

3.5 Guidelines for Effective Disclosures

The above discussion of dimensions for educational disclosures presents a complex picture of the tradeoffs inherent in the design of a self-disclosing system. Note that the design decisions are even more complicated when one considers the potential for interactions between dimensions. For example, the adaptivity of a self-disclosing system could influence the ability of a tool to present relevant disclosures or structure them appropriately. Furthermore, the intrusiveness of a system could affect the scope of disclosures, since the larger the screen space devoted to disclosures the greater the variety of feedback that could be shown at once. On the other hand, the evidence presented above from learner-centered educational design and human language acquisition suggests a number of ways in which the self-disclosure design space could be simplified to ensure pedagogically effective systems. A useful way of viewing these simplifications is as constraints on the ranges of degrees along each disclosure dimension as shown in Figure 3-3. For adaptivity and explorability, educational computing and language acquisition work suggests a design emphasizing the "high" end of the spectrum. In the case of intrusiveness and structure, previous research predicts that systems between the low and intermediate range of the spectrum will have a positive educational impact. The two recommended ranges along the relevance dimension in Figure 3-3 reflect two themes in research on contextualized learning: (a) that low relevance can support serendipity, and (b) that high relevance feedback is particularly memorable. Finally, there is little evidence to indicate constraints on the scope dimension, although common sense suggests one avoid the extreme low end (disclosing nothing) and the extreme high end (disclosing everything).

Let us now return to the four examples of existing self-disclosing systems and compare them to the above profile of an educationally effective one. Figure 3-4 indicates an estimate of where each of the four systems falls along each of the six disclosure dimensions.

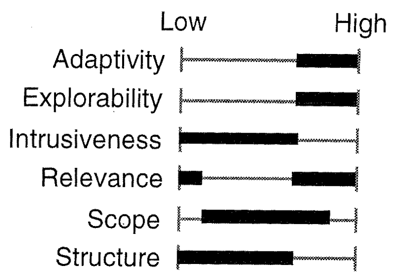


Figure 3-3. The ranges of degrees along disclosure dimensions that are likely to be most pedagogically effective, particularly for teaching programming.

Navigator Status Line. By being neither adaptive nor explorable Netscape Navigator's status line fails to meet the guidelines for the first two dimensions. However, by using only a small of amount of screen space at the bottom of its windows the status line does fall under the suggested range of intrusiveness. The internal connection information displayed by the status line overlaps the suggested degrees of relevance, since it can sometimes provide the user with potentially important information about the current status of the network. The scope of the status line disclosures is limited to a small but reasonable subset of possible connection information. Finally, the sequence of Navigator's feedback is simply dictated by the connection protocol, not by any curriculum, therefore the system falls on the low end of the structure spectrum, but still within the suggested profile.

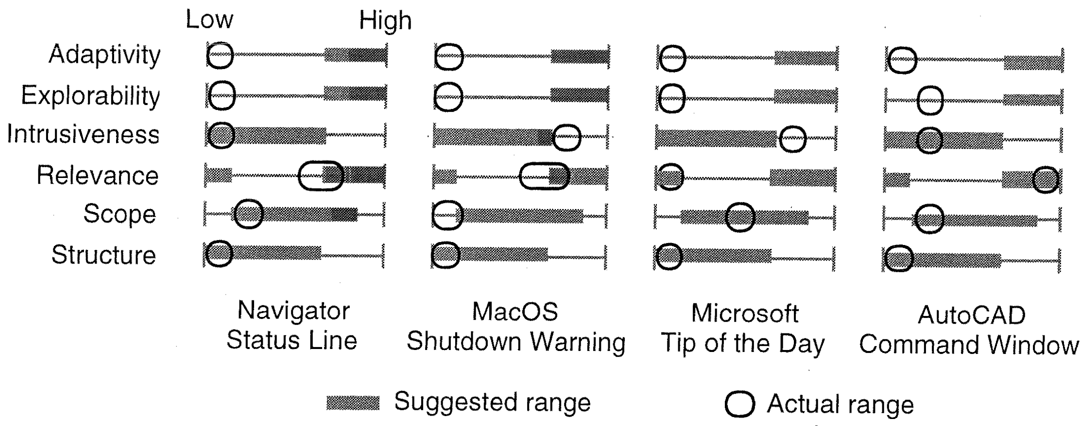


Figure 3-4. A comparison of four existing self-disclosing systems using the six dimensions of educational disclosures.

Mac OS Shutdown Warning. Unlike Navigator, the Mac OS shutdown warning is beyond the suggested range of intrusiveness by requiring the user to explicitly dismiss the dialog. The Mac OS disclosure overlaps the suggested ranges of relevance, since it can sometime provide timely information about how to properly shut down, assuming the user starts the computer soon after unplugging. Obviously, being only a shutdown warning, the scope and structure of this Mac OS disclosure is extremely limited.

Microsoft Tip of the Day. As a modal dialog box, Microsoft's Tip of the Day shares some of the disclosure characteristics of the Mac OS shutdown warning. By displaying random hints, the system falls on the low end of the relevance spectrum, at the point where serendipity might be advantageous. Unlike the shutdown warning, Tip of the Day disclosures offer a broad scope of suggestions, although typically limited to advanced features.

AutoCAD Command Window. AutoCAD is the only self-disclosing system of the four that enables users to experiment with and discover more about its disclosures. Since exploration is limited to only evaluating a disclosed command language expression, AutoCAD's form of self-disclosure still falls short of the recommended range. By echoing equivalent expressions in response to the use of direct manipulation tools, AutoCAD achieves a high degree of relevance, well within the suggested profile. The scope of AutoCAD disclosures is limited to only those direct shape creation commands that have command line equivalents. Finally, AutoCAD makes no attempt to sequence disclosures for the pedagogical purposes and so falls on the low end of the recommended range of structure.

4. Chart ‘n’ Art: A Prototype Self-Disclosing Tool

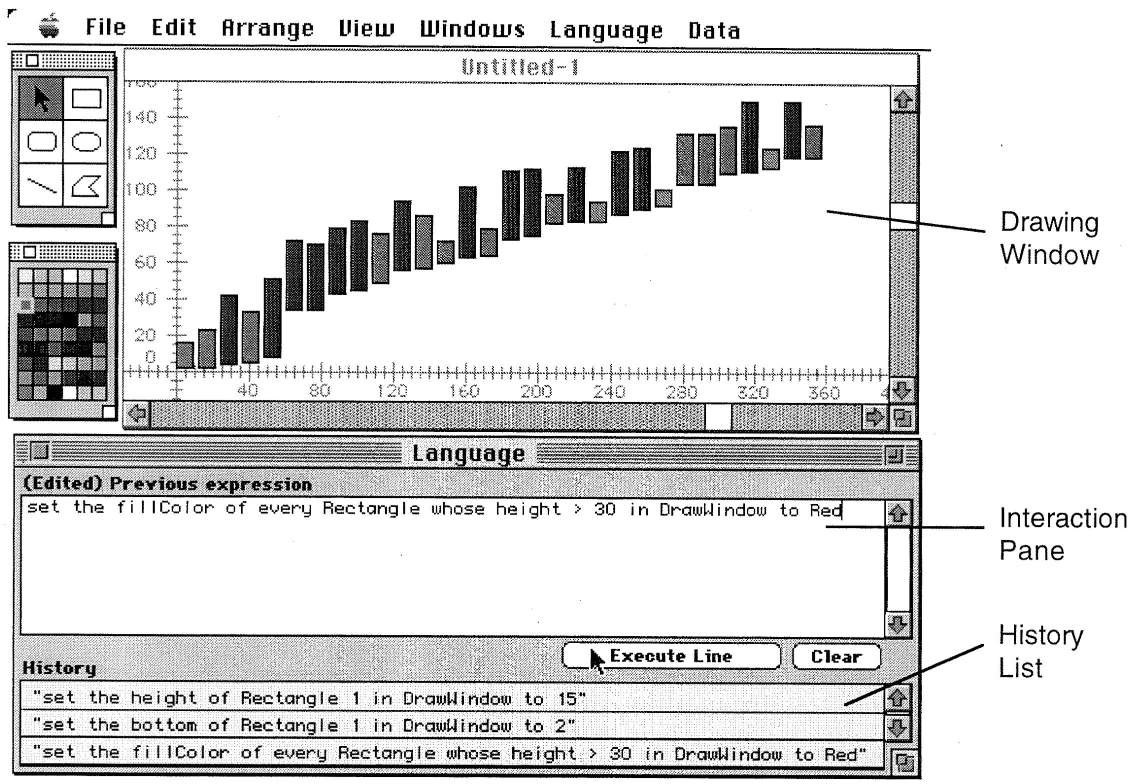
Chart ‘n’ Art is a programmable information graphics tool developed by the author to explore a variety of self-disclosure mechanisms and to provide a vehicle for evaluating the educational effects of disclosures on users. This system is by no means the ultimate self-disclosing tool, but rather represents a region in the self-disclosing design space characterized in Chapter 3. This chapter describes drawing, programming, learning, and exploring with Chart ‘n’ Art. The next chapter will present various perspectives on the system including its implementation, the evolution of its interface, and its relationship to the self-disclosure design space and existing tools. Readers seeking more detailed information on Chart ‘n’ Art and its language should consult the *Chart ‘n’ Art User Manual* (DiGiano, in preparation) as well as the *SK8 User Guide* (Apple Computer Inc., 1996b).

4.1 Drawing with Chart ‘n’ Art

At first glance, Chart ‘n’ Art looks like a standard Mac OS direct manipulation drawing package such as MacDraw, with a tool palette, color palette, drawing window and menus as shown in Figure 4-1. With its self-disclosure mechanisms disabled, the system, in fact, behaves like a simple drawing package. Users create a shape by selecting a tool from the palette and then dragging the mouse to form the outline of the desired shape in the drawing window. Creating a polygon requires that users click the mouse to indicate vertices; a double click closes the polygon. The currently selected item in the color palette determines the interior color of new shapes. A new color can be assigned to a shape by selecting the object and then selecting a new item in the color palette. Chart ‘n’ Art does not currently allow users to change the border color of shapes via direct manipulation.

Shapes can be resized by selecting the item in the drawing and dragging one of eight small black “handles” which appear on the shape’s perimeter. Shapes can be moved by dragging the item to a new position. Holding down the shift key constrains the movement to either the horizontal or vertical plane depending on the direction of the initiating drag gesture. Users can also rearrange shapes via menu commands. Layering menu commands change the stacking order of shapes. Distribute menu commands move selected shapes so that there is equal distance between their centers either measured horizontally, vertically, or diagonally.

Chart ‘n’ Art provides the standard Mac OS Edit menu commands for cutting, copying, and pasting shapes. There are two additional menu commands to facilitate copying items: The first is the Duplicate menu command that makes a single copy of the selected shape or shapes and offsets the new shapes above and to right of their originals. The second is the Duplicate Special menu command that brings up a dialog box (Figure 4-2) where users can specify an arbitrary number of desired copies and the exact horizontal and vertical offsets to be used for distributing the duplicates.



(Edited) Previous expression

set the fillColor of every Rectangle whose height > 30 in DrawWindow to Red

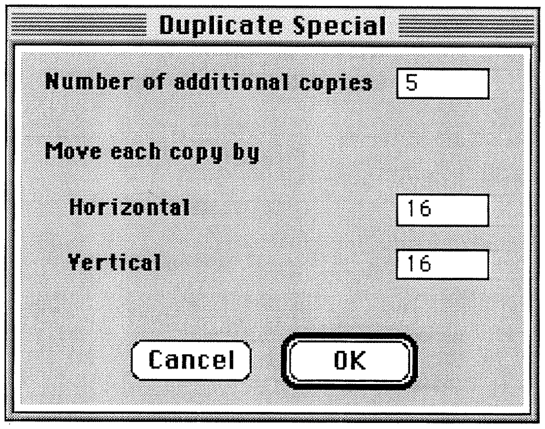
History

"set the height of Rectangle 1 in DrawWindow to 15"

"set the bottom of Rectangle 1 in DrawWindow to 2"

"set the fillColor of every Rectangle whose height > 30 in DrawWindow to Red"

Figure 4-1. Chart 'n' Art's basic windows, palettes and menus. Except for the Language window at the bottom, the system looks like a basic drawing package such as MacDraw.



Number of additional copies: 5

Move each copy by

Horizontal: 16

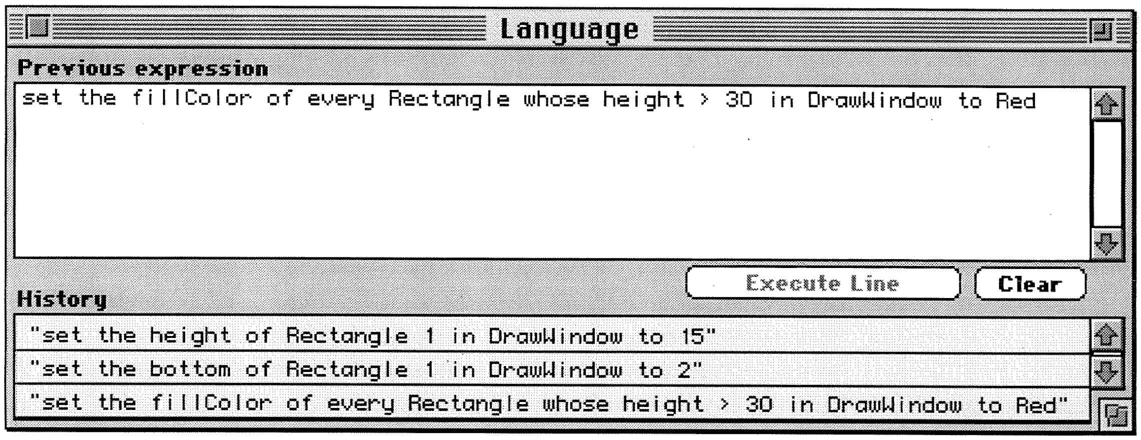
Vertical: 16

Figure 4-2. The Duplicate Special dialog box used for specifying numbers of desired copies and the offsets for distributing the new shapes.

4.2 Programming with Chart 'n' Art

Like its predecessors SchemePaint (Eisenberg, 1995) and SchemeChart (Eisenberg & Fischer, 1994), Chart 'n' Art is a programmable application with programming language features that complement direct manipulation operations. Unlike its predecessors, Chart 'n' Art's architecture allows it to be programmed through a variety of languages, making it possible to compare and contrast the learnability of various end-user programming languages. Currently only SK8Script and Lisp disclosures are supported. As mentioned earlier, SK8Script is an interpreted, object-oriented, English-like language related to AppleScript and HyperTalk. Although SK8Script and Lisp have a very different syntax, the operations typically used in Chart 'n' Art are specified with similar keywords in both languages. For simplicity's sake, this report will present examples only in the SK8Script language and illustrate system interaction with the SK8Script interpreter.

Users can edit and evaluate SK8Script expressions in the Language window shown in Figure 4-3. Typing an expression into the Interaction pane of the Language window and clicking the Execute Line button (or pressing the Enter key) causes that expression to be evaluated. Previously executed expressions accumulate in the history list at the bottom of the window. These history entries can be copied and edited or combined to form new expressions. Selecting a history entry automatically copies the text of the previous expression into the Interaction pane for reuse. Selecting multiple history entries sends a concatenation of their corresponding expressions to the Interaction pane.



```
set the fillColor of every Rectangle whose height > 30 in DrawWindow to Red
```

```
"set the height of Rectangle 1 in DrawWindow to 15"
"set the bottom of Rectangle 1 in DrawWindow to 2"
"set the fillColor of every Rectangle whose height > 30 in DrawWindow to Red"
```

Figure 4-3. The Language window where users can view disclosures and enter expressions to be executed.

4.2.1 Chart 'n' Art's Enriched SK8Script

The programming language available to Chart 'n' Art users is actually an enriched version of SK8Script with extra facilities for manipulating graphical objects. There are also a few modifications to standard SK8Script to support the most typical graphics operations. These additional language features were kept to a minimum, since the goal of Chart 'n' Art was to explore mechanisms for improving the learnability of the existing language, not enhancing the design of the language itself.[6]

To understand how one uses SK8Script for managing drawings, it helps to be familiar with the language's underlying object model called the SK8 Object System. This object system is prototype-based, meaning that any object can be a template for creating new objects, obviating the need for object classes. Objects encapsulate data in the form of "properties" and procedures in the form of "handlers." To facilitate graphics operations, the SK8 Object System contains "Actor" objects which have properties for position, color, size, etc. and handlers for drawing themselves, dealing with mouse and keyboard input, and so on.

[6] For an in depth discussion of the potential power of well-designed domain-enriched languages see (Eisenberg, 1995).

The direct manipulation tools in Chart 'n' Art's drawing palette are in effect creating new instances of Actors and installing them into the drawing window. Alternatively, users can perform the same operation with Chart 'n' Art's special programming language function, `make`. Make creates a new Actor of a given type and installs it into the front-most drawing window (unless a container is specified). Users can specify the initial properties of the new actor using '`with`' clauses. For example, the following expression creates a new line and provides its initial end points:

```
make LineSegment with head {20, 30} with tail {100, 115}
```

Note that coordinates in SK8Script are simply lists of two values with lists delimited by curly brackets and each item within the list separated by commas. If users do not indicate the position of a new Actor, the default location is the center of the visible part of the drawing window. If no color is specified, the default fillColor is the currently selected item in the color palette. Make currently can create instances of the following SK8 Actors: Rectangle, RoundRect, Oval, LineSegment, and Polygon, corresponding to each of the shapes that appear in Chart 'n' Art's tool palette.

Once Actors are installed in a Chart 'n' Art drawing window, users can refer to one or more of the shapes via SK8Script in a variety of ways. The most common method is to use what is called a "selection expression." Selection expressions provide a powerful, consistent way to identify a single object or a "collection" of objects. Selection expressions include a *source*, *filter*, *selector*, and *preposition*. The *source* can be any expression that evaluates to a collection of objects, although most often in Chart 'n' Art it is the front-most drawing window, referenced by the global variable `DrawWindow`.[7] When the source of a selection expression is a window, only the Actors within that window are searched for a match. The *filter* limits such a search to objects that are of a specific type or objects that pass an arbitrary test. The *selector* chooses a particular item or subset of items from the filtered source. Finally, the choice of *preposition* can provide added guidance for the selection, although for most Chart 'n' Art applications only the keyword '`in`' is used.

Consider the following selection expression:

```
Rectangle 2 in DrawWindow
```

In this example, the source is DrawWindow, the front-most Chart 'n' Art drawing. The filter is the type filter `Rectangle`, so the search is limited to only rectangle shapes in the drawing. The selector is the index 2; that is, it selects the second object returned by the search. The result is a reference to the second rectangle drawn by the user in the front-most drawing window. If, instead, one wanted a collection of all the rectangles in the drawing, the selector `every` could be used:

```
every Rectangle in DrawWindow
```

The keyword 'every' causes the selection expression to return a list of all the members of the source that match the filter. If one wanted all the shapes in the drawing, the type filter could be changed to the wildcard keyword '`item`':

[7] Technically, the DrawWindow variable does not refer to a window, but an instance of invisible Actor within a drawing window called 'PaperSurface.' Shapes drawn via direct manipulation or the 'make' function are installed so that their direct container is a PaperSurface, not a window. The reason this is done is that drawing windows contain extra actors such as scroll bars and axes that would complicate selection expression. A PaperSurface contains no such clutter, only the shapes that users might want to search.

```
every item in DrawWindow
```

SK8Script also supports test filters made up of Boolean expressions:

```
every item whose fillColor = Red in DrawWindow
```

This selection expression returns all of the shapes in the drawing that are red.

Selection expressions in Chart 'n' Art are particularly useful in combination with the assignment operator, `set`. The `set` operator has a number of variants. The simplest stores a value in a variable:

```
set x to 15
```

But set can also be used to change the value of an object's properties:

```
set the fillColor of Oval 3 in DrawWindow to Green
```

Here, `set` takes a property name, `fillColor`; a target object—in this case the result of the selection expression "`Oval 3 in DrawWindow`"; and a new value, `Green`. This is the most common technique for making changes to a shape in Chart 'n' Art. A similar assignment statement can be used to change the attributes of more than one shape. For instance,

```
set the bottom of every Rectangle in DrawWindow to 0
```

will make the bottom edge of each rectangle in the drawing have a vertical position of 0, thereby aligning the bottoms of all of the rectangles. In this document, such an assignment statement is called a "one-to-many mapping." There is also a "many-to-many mapping" which for a particular property assigns a list of values in turn to a list of objects. In SK8Script this type of assignment is indicated by the words "every item in" followed by a list of values at the end of the statement:[8]

```
set the bottom of every Rectangle in DrawWindow to every item in {0,
10, 20, 30}
```

In this example, the bottom edge of the first four rectangles in the drawing are assigned to the vertical positions 0, 10, 20, 30 respectively.

SK8Script's one-to-many and many-to-many mapping expressions provide a powerful and succinct way to specify iteration implicitly. These expressions enable users to automate a surprisingly large number of drawing tasks with Chart 'n' Art. However, there are also occasions for explicit iteration. Most explicit iteration is accomplished through the `repeat` command which comes in a number of forms, the most common of which appear in Table 4-1.

4.2.2 An Example

Chart 'n' Art's language facilities are what make the system particularly suited for information graphics. To illustrate the utility of programming with Chart 'n' Art let us return to the *Daily Camera* case study in Chapter 2. Recall that a designer at the *Camera* is responsible for generating a floating bar chart each month to depict daily temperature ranges from the previous month. In actuality the data for the temperature chart consists of highs and lows for each day in the month, but to keep things simple for now let us

[8] Many-to-many mappings using a single assignment statement are available in SK8 version 0.91 and earlier. The forthcoming version SK8 1.0 will not have this feature. Instead, users will be required to nest an assignment statement within a loop.

Table 4-1. Some of the more common forms of explicit in SK8Script using the `repeat` *command.*

Application	Example

perform commands a fixed number of times	`repeat 5 times`
	`   beep`
	`end repeat`

iterate over a sequence of numeric values	`repeat with i from 2 to 5`
	`   set the fillColor of Oval i to Red`
	`end repeat`

perform commands for each item in a collection	`repeat with shape in DrawWindow's selection`
	`   set the left of shape to 20`
	`end repeat`

suppose the data is lows and ranges for each day. For example, instead of being given a list of highs and a list of lows such as

```
{17, 3, 18, 57, ...}
{-6, -16, -13, 0, ...}
```

suppose that the designer instead received the following two lists:

```
{-6, -16, -13, 0, ...}
{23, 19, 31, 57, ...}.
```

As an additional simplification, let us assume that the designer has already constructed an appropriate set of axes that can be pasted into the back layer of the chart.

As a final note, SK8Script programmers should be aware that the language has been altered slightly to accommodate the drawing needs of Chart 'n' Art users. One important change is that Chart 'n' Art shapes use a standard Cartesian coordinate system, not a "screen coordinate" system. This means that objects that are closer to the top of the screen than other objects have a greater y coordinate. For technical reasons, most Actor properties which are dependent on screen coordinates are not reimplemented in Chart 'n' Art to work with mathematical coordinates. Instead, Chart 'n' Art Actors have alternative properties which work with mathematical coordinates as listed in Table 4-2. Users can refer to all other Actor properties such as `height`, `width`, `left`, `right`, `top`, and `bottom` with the original SK8Script names.

The central task for the designer then, is to place and size the appropriate number of temperature range bars to go on top of the axes. A generally effective strategy in Chart 'n' Art is to produce all of the graph elements either by direct manipulation or by SK8Script and then adjust them for their exact values later. This allows the users to take advantage of the implicit iteration in many-to-one or many-to-many mapping expressions. In the case of the temperature chart, the designer might initially create the bars by either (a) dragging out the shape of a model bar and using the Duplicate Special menu command (see Figure 4-2) to create the other necessary bars, or (b) evaluating an iteration expression which generates the bars automatically. There are a number of programming solutions for this second option. For instance, for the month of February 1996, the designer could use the following expression to generate 29 6-unit-wide rectangles, one for each day in the month (1996 is a leap year):

Table 4-2. Some mathematically-based coordinate properties of Chart 'n' Art Actors. These properties are distinguished from the standard Actor properties by the fact that they refer to a coordinate system where y-values increase going up.

Property	Applicable Actors	Description
`bounds`	all	A four item list denoting the bounding rectangle of a shape. (Typically {left, top, right, bottom})
`center`	all	A two item list denoting the Cartesian coordinate of the middle of a shape.
`head`	LineSegment	A two item list denoting the Cartesian coordinate of the starting point of a line segment.
`tail`	LineSegment	A two item list denoting the Cartesian coordinate of the ending point of a line segment.
`vertices`	Polygon	An evenly-sized list consisting of alternating x- and y-coordinates for each point in a polygon.

```
repeat 29 times
   make Rectangle with width 6
end repeat
```

The designer would then need to use the Distribute menu command or the distributeHorizontally function to spread out the rectangles along the horizontal axis. The result would look like Figure 4-4a. A somewhat more complicated iteration expression could distribute the new rectangles automatically:

```
repeat with i from 1 to 29
   make Rectangle with bounds {(i-1) * 8, 50, (i-1) * 8 + 6, 0}
end repeat
```

In this case, each rectangle is created with a slightly different `bounds` property where the `bounds` is a list of four values denoting the left, top, right, and bottom of the shape. Since the left and right values are dependent on the loop variable '`i`', rectangles would be placed with successively increasing horizontal positions, thereby distributing them across the horizontal axis automatically.

The next step is for the designer to adjust the vertical position of each bar to correspond to the lows data for the month of February. Using SK8Script's many-to-many mapping expressions, this requires only a single statement (assuming a 1:1 scale):

```
set the bottom of every Rectangle in DrawWindow to ¬
   every item in {-6, -16, -13, 0, 42, 45, 35, 51, 41, ¬
      29, 19, 25, 27, 39, 25, 17, 28, 45, 37, 39, 38, ¬
      34, 24, 30, 29, 17, 11, -3, 2}
```

The result would look like Figure 4-4b. Finally, the designer could use the same many-to-many mapping structure to stretch the bars to their proper size with respect to the range data for the month of February:

```
set the height of every Rectangle in DrawWindow to ¬
   every item in {23, 19, 31, 57, 13, 14, 25, 13, 29, 21, ¬
      37, 35, 39, 21, 26, 50, 37, 16, 24, 24, 25, 29, ¬
      26, 37, 32, 8, 10, 24, 35}
```

The end result appears in Figure 4-4c.

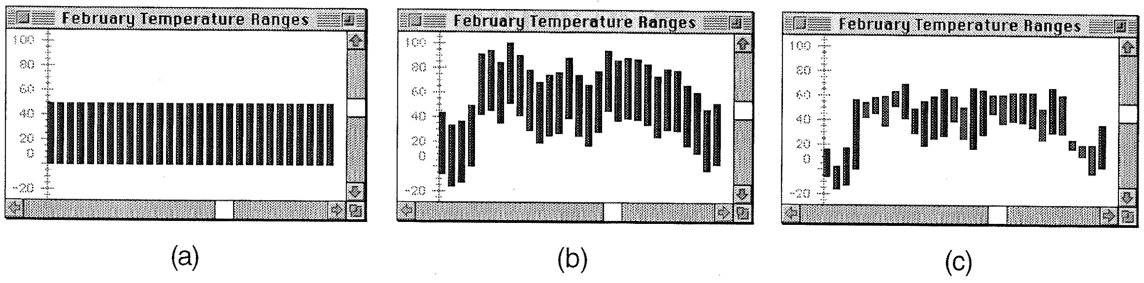


Figure 4-4. Steps in creating The Daily Camera's *temperature chart using programming in Chart 'n' Art. Compare the result in (c) to the actual chart printed by the* Camera *which is reproduced in Figure 2-4.*

The entire process for creating the temperature chart could be further simplified through procedural abstraction. In SK8Script, the designer could declare a new function as follows:

```
on temperatureChart of lowList, rangeList
   repeat with i from 1 to the length of lowList
      make Rectangle with bounds {(i-1) * 8, 50, (i-1) * 8 + 6, 0}
   end repeat
   set the bottom of every Rectangle in DrawWindow to ¬
      every item in lowList
   set the bottom of every Rectangle in DrawWindow to ¬
      every item in rangeList
end temperatureChart
```

Certain details have been left out of this example programming solution including scaling and axes creation. Certainly, a commercial programmable charting tool would need primitives for dealing with these issues. However, this example does illustrate key techniques useful for a variety of charting tasks. Many of these techniques could be used in the charting tasks highlighted in the three other case studies described in Chapter 2.

4.3 Learning with Chart 'n' Art

Although the language-based solution in the previous section illustrates the potential for programming information graphics, Chart 'n' Art was developed not so much to support programming in this context, but to investigate self-disclosure mechanisms for helping users learn a programming language in the first place. As shown in Table 4-3, there are four basic self-disclosure mechanisms in Chart 'n' Art: mouse disclosures, selection labeling, alternatives disclosures, and possibility disclosures. The self-disclosure techniques illustrate the different roles for disclosures, use a variety of presentational methods, are triggered by a number of different events, and are designed to teach various types of programming knowledge. The goal was to transform almost every form of user interaction with Chart 'n' Art into a potential opportunity for learning programming.

This section describes each of Chart 'n' Art's four self-disclosure mechanisms in turn. To illustrate how the mechanisms work and what can be learned from them, this section presents hypothetical interactions between Chart 'n' Art and a fictitious designer named Alice. Alice works for *The Daily Camera* and among other things is responsible for creating the monthly temperature chart. In these hypothetical situations Alice will initially employ the actual direct manipulation techniques currently used at the Daily

Camera to produce the temperature charts. These real-life techniques are described more fully in Chapter 2.

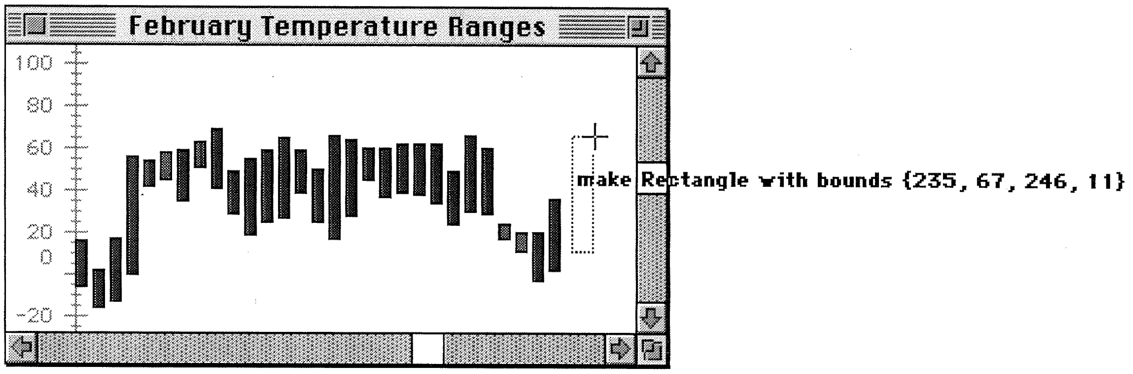
Table 4-3. Characteristics of the four basic self-disclosure mechanisms in Chart 'n' Art, including the target programming knowledge each is intended to teach. (d.m. stands for "direct manipulation.")

	Mouse Disclosures	Selection Labeling	Alternatives Disclosures	Possibility Disclosures
Role	echoing monitoring	identifying	echoing	advertising
Presentation	mouse text	selection label	language window	language window
Trigger	mouse movement	selection	d.m. completion	selection
Target Programming Knowledge	shape properties, assignment, coordinate system	selection expressions, test filters, referential alternatives	functional alternatives, nesting, sequencing	properties inaccessible by d.m., mapping expressions

4.3.1 Mouse Disclosures

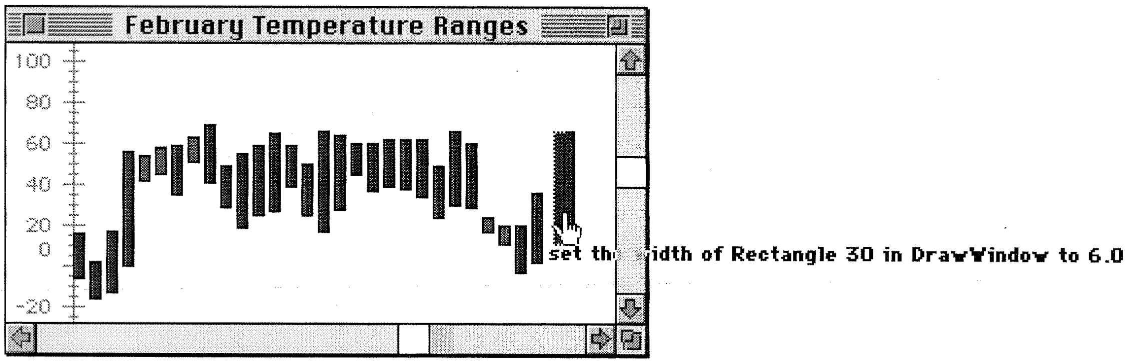
A mouse disclosure in Chart 'n' Art is a single line of text that appears just below and to the right of the cursor as users create and manipulate shapes by dragging the mouse as shown in Figure 4-5. This text is a SK8Script expression that users could evaluate in place of the action they are currently performing by direct manipulation (an example of *echoing*). Disclosures include `make` expressions for generating new shapes, `set` expressions for assigning new values for a shape's dimensions, and `move` expressions for adjusting the position of one or more shapes. These expressions often contain references to a shape's coordinates, either its four item `bounds` list, or its two item `center` list (see Table 4-2.) As users drag a shape's outline, these coordinates are continuously updated (an example of *monitoring*). Such feedback provides users with a convenient guide for accurately positioning or sizing shapes. The educational goal of mouse disclosures, however, is to expose users to the shape property vocabulary, the syntax for assigning values to these properties, and the Chart 'n' Art coordinate system.

To illustrate mouse disclosures, suppose Alice is producing a temperature chart for the month of March. Unfortunately, as is sometimes the actual case, Alice cannot locate March's file from the previous year, so she must modify a different month, say, February. Now since March has more days than February, Alice must first add extra bars to the chart. As she drags the outline of a new bar, a `make` mouse disclosure appears by her cursor as shown in Figure 4-5a, indicating the expression she could use to generate the shape by programming instead of direct manipulation. The four coordinates in the mouse disclosure update continuously as she drags the mouse. The new bar is too wide, so Alice next drags the shape's middle left handle to make it narrower, causing the 'set' mouse disclosure in Figure 4-5b to appear by her cursor.



make Rectangle with bounds {235, 67, 246, 11}

(a)



set the width of Rectangle 30 in DrawWindow to 6.0

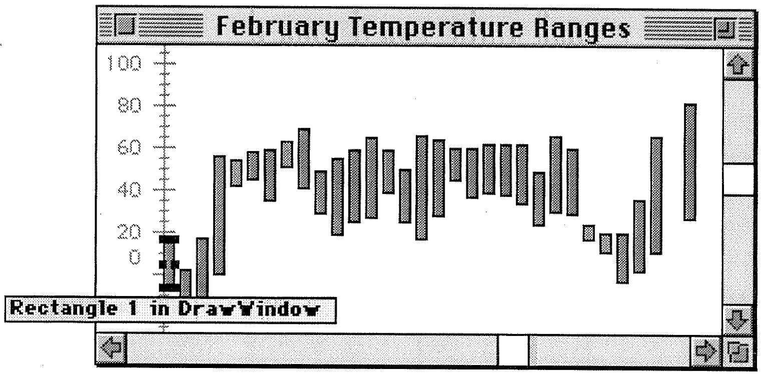
(b)

Figure 4-5. (a) A `make` *mouse disclosure which appears while Alice is dragging the outline of a new bar. (b) A* `set` *mouse disclosure which appears while Alice is adjusting the width of the new bar. The mouse text reads, "set the width of Rectangle 30 in DrawWindow to 6.0"*

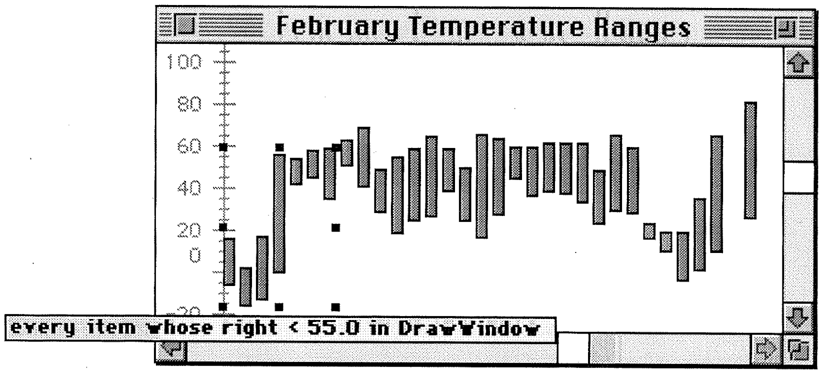
4.3.2 Selection Labeling

A selection label is a line of identifying text that appears just below the shape or shapes currently selected in a Chart 'n' Art drawing window as shown in Figure 4-6. The text is a valid SK8Script expression for referring to the active selection. For a single item selection what appears is a simple selection expression (see Section 4.2.1) using a type filter and index, such as "`Oval 3 in DrawWindow`." Multiple-item selections can have a variety of labels from "`every item in DrawWindow`" (when all shapes are selected) to more complex selection expressions using type filters and/or test filters. By default, the system will first try and use a simple type filter (e.g. "`every Oval in DrawWindow`"); if this is not possible it will try a test filter using color (e.g. "`every item whose fillColor = Blue`"), and then test filters referring to other properties. When a selection expression cannot be found for multiple items, Chart 'n' Art simply labels them with a list expression containing a selection expression for each individual item. A selected group of shapes requiring a long list label is instead given the label "`every item in the selection of DrawWindow`." The educational goals of selection labeling is to (a) familiarize users with the SK8Script selection expression syntax in general, (b) expose users to particular test filters that provide a powerful means of searching shapes, and (c) make users aware of the variety of ways to refer to objects with the programming language.

Figure 4-6 depicts selection labeling just after Alice has selected the bar representing the first day of the month and then after selecting the first seven days in the month. The



(a)


February Temperature Ranges
100
80
60
40
20
0
every item whose right < 55.0 in DrawWindow

(b)

Figure 4-6. (a) A selection label appearing beneath Alice's current single-item selection. (b) A selection label referring to the first seven bars in the chart.

latter selection label is a selection expression with a test filter that searches for shapes to the left of 55 on the horizontal axis.

4.3.3 Alternatives Disclosures

An alternatives disclosure in Chart 'n' Art is a set of equivalent expressions that appear in the Interaction pane of the Language window after a direct manipulation action is completed. Similar to mouse disclosures, these expressions provide options for accomplishing the same operation via SK8Script instead of the mouse. But since alternatives disclosures occupy a separate scrollable window pane, their role is to expand on mouse disclosures and provide additional language feedback. Disclosures include the `make`, `set`, and `move` expressions in mouse disclosures and also more sophisticated expressions involving mapping, iteration, nested statements, sequenced statements, and temporary variables. Aside from exposing users to these specific types of expressions, the educational intent of alternatives disclosures is to make users aware that (a) there is often more than one way to accomplish a task using programming, and (b) expressions can be composed into arbitrarily complex statements by nesting statements and by sequencing multiple lines of code.

To illustrate alternatives disclosures, let us return to Alice's last direct manipulation action in Section 0. The moment Alice lets go of the mouse to fix the width of her new bar, the language window displays an alternatives disclosure consisting of two alternative expressions as shown in Figure 4-7. Below each SK8Script alternative is explanatory text in italics, which is typically automatically generated from a language definition file, although custom explanations are possible. Suppose Alice continues adapting the February chart for March and adds the final bar by copying the last one she just drew with the mouse using the menu command Duplicate. This triggers the following alternatives disclosure illustrating nesting, sequencing, and temporary variables:

```
move copy Rectangle 30 in DrawWindow by {16, 16}
-- Makes a copy of Rectangle 30 in DrawWindow and returns the duplicate.
-- Then offsets the position of that result by {16, 16}.

set dup to copy Rectangle 30 in DrawWindow
move dup by {16, 16}
-- Makes a copy of Rectangle 30 in DrawWindow and returns the duplicate.
-- Then assigns the variable dup to that result.
-- Offsets the position of dup by {16, 16}.
```

Alternatives disclosures showing iteration and mapping are also possible. If, say, Alice decided to change the colors of all the bars by selecting them and clicking on light green in the color palette, she would see the following alternatives in the language window:

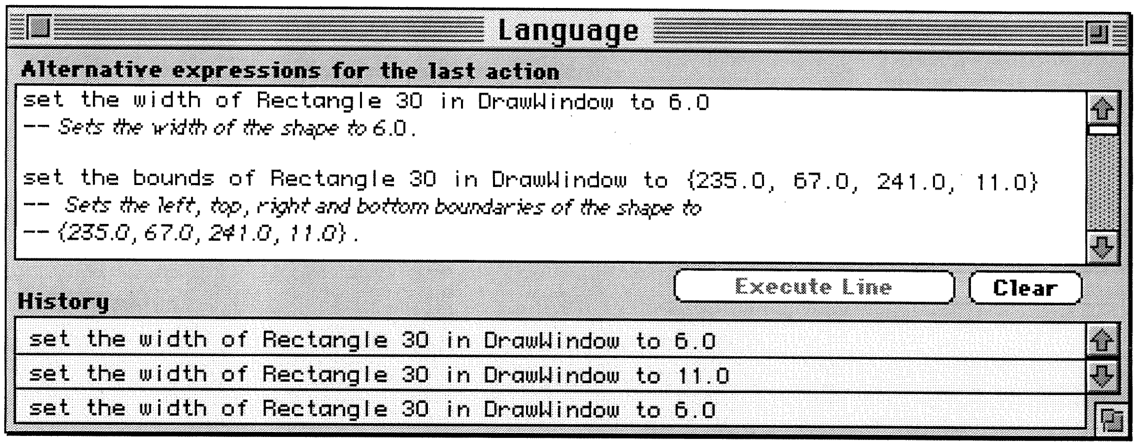
```
set the fillColor of every item in DrawWindow to LightGreen
-- Sets each shape's interior color to LightGreen.

set the fillColor of every item in DrawWindow to ¬
    every item in {LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen, LightGreen, ¬
                   LightGreen, LightGreen, LightGreen}
-- Sets each shape's interior color to each item in {LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen,
-- LightGreen, LightGreen, LightGreen, LightGreen, LightGreen},
--  respectively.

repeat with i in every item in DrawWindow
   set the fillColor of i to LightGreen
end repeat
-- Sets  to LightGreen. for each element in each item in DrawWindow
```

4.3.4 Possibility Disclosures

A possibility disclosure is a list of expressions that appear in the Interaction pane of the Language window after users select one or more shapes with the mouse. This type of *advertising* disclosure describes possible ways in which users could potentially operate on the selected shapes using programming. This disclosure mechanism is designed to expose users to expressions that might appear infrequently, if at all, in other forms of feedback such as mouse disclosures or alternatives disclosures. Indeed, many operations in Chart 'n' Art such as setting the attributes of a shape's border (`frameColor` and `frameSize`) or the weight of a line segment (`lineSize`) are currently possible only through programming. Therefore expressions like these, which would never be echoed as a result of direct manipulation, can still be revealed to users. In



```
set the width of Rectangle 30 in DrawWindow to 6.0
-- Sets the width of the shape to 6.0.

set the bounds of Rectangle 30 in DrawWindow to {235.0, 67.0, 241.0, 11.0}
-- Sets the left, top, right and bottom boundaries of the shape to
-- {235.0, 67.0, 241.0, 11.0}.
```

```
set the width of Rectangle 30 in DrawWindow to 6.0
set the width of Rectangle 30 in DrawWindow to 11.0
set the width of Rectangle 30 in DrawWindow to 6.0
```

Figure 4-7. An alternatives disclosure in response to Alice narrowing one the bars with the mouse. Note that the Interaction pane label changes to indicate that the text below contains, "alternative expressions for the last action."

addition to describing properties inaccessible by direct manipulation, a possibility disclosure on a group of shapes reveals one-to-many and many-to-many mapping expressions for adjusting the group's attributes *en masse*. The choice of selection expression used to refer to such a group of objects is determined by the same set of rules used for selection labeling. Like alternatives disclosures, expressions generated by a possibility disclosure are each annotated with explanatory English text.

Let us return to Alice's temperature chart to describe possibility disclosures in action. Suppose Alice now has all of the bars she needs for March. Her next task is to adjust their bottoms and tops for the March highs and lows data. Recall that the technique actually used at the Daily Camera is to select each bar, one at a time, and drag its ends to the appropriate vertical positions. When Alice selects the first bar in Chart 'n' Art, the following appears in her Language window's Interaction pane (explanations have been removed to conserve space):

```
set the frameColor of Rectangle 1 in DrawWindow to Black

set the fillColor of Rectangle 1 in DrawWindow to LightGreen

set the frameSize of Rectangle 1 in DrawWindow to {1.0, 1.0}

set the bottom of Rectangle 1 in DrawWindow to -6.0

set the right of Rectangle 1 in DrawWindow to 6.0

set the top of Rectangle 1 in DrawWindow to 17.0

set the left of Rectangle 1 in DrawWindow to 0.0

set the width of Rectangle 1 in DrawWindow to 6.0

set the height of Rectangle 1 in DrawWindow to 23.0

set the center of Rectangle 1 in DrawWindow to {3.0, 5.5}

set the bounds of Rectangle 1 in DrawWindow to {0.0, 17.0, 6.0, -6.0}
```

Note that each expression in the disclosure demonstrates how to operate on a different property of the selected rectangle. The values being assigned to each property represent the current state of the selected shape. In other words, evaluating any of the

```
set the frameColor of every item whose right < 55.0 in DrawWindow to Black

set the fillColor of every item whose right < 55.0 in DrawWindow to ¬
    LightGreen

set the frameSize of every item whose right < 55.0 in DrawWindow to ¬
    {1.0, 1.0}

set the bottom of every item whose right < 55.0 in DrawWindow to ¬
    every item in {-6.0, -16.0, -13.0, -0.0, 42.0, 45.0, 35.0}

set the right of every item whose right < 55.0 in DrawWindow to ¬
    every item in {6.0, 14.0, 22.0, 30.0, 38.0, 46.0, 54.0}

set the top of every item whose right < 55.0 in DrawWindow to ¬
    every item in {17.0, 3.0, 18.0, 57.0, 55.0, 59.0, 60.0}

set the left of every item whose right < 55.0 in DrawWindow to ¬
    every item in {0.0, 8.0, 16.0, 24.0, 32.0, 40.0, 48.0}

set the width of every item whose right < 55.0 in DrawWindow to 6.0

set the height of every item whose right < 55.0 in DrawWindow to ¬
    every item in {23.0, 19.0, 31.0, 57.0, 13.0, 14.0, 25.0}

set the center of every item whose right < 55.0 in DrawWindow to ¬
    every item in {{3.0, 5.5}, {11.0, -6.5}, {19.0, 2.5}, {27.0, 28.5}, ¬
                   {35.0, 48.5}, {43.0, 52.0}, {51.0, 47.5}}

set the bounds of every item whose right < 55.0 in DrawWindow to ¬
    every item in {{0.0, 17.0, 6.0, -6.0}, {8.0, 3.0, 14.0, -16.0}, ¬
                   {16.0, 18.0, 22.0, -13.0}, {24.0, 57.0, 30.0, -0.0}, ¬
                   {32.0, 55.0, 38.0, 42.0}, {40.0, 59.0, 46.0, 45.0}, ¬
                   {48.0, 60.0, 54.0, 35.0}}
```

Figure 4-8. The possibility disclosure after Alice selects the first week's worth of bars in the temperature chart.

expressions immediately following a possibility disclosure should have no effect on the current state of the drawing window.

A more interesting possibility disclosure appears when Alice selects the first week's worth of bars, as shown in Figure 4-8. In this case, Chart 'n' Art discloses the same set of properties, but the simple assignment statements are replaced with one-to-many and many-to-many mapping expressions operating on all seven bars. The one-to-many mappings represent properties such as `fillColor` for which the seven bars share the same value. Many-to-many mappings, on the other hand, denote properties such as `height` for which the seven bars have different values. Note that the list of values in the many-to-many mapping for the `bottom` property matches the list of low temperatures for the first week in February.

4.4 Experimenting with Chart 'n' Art

The disclosures described in the previous section are intended not only to offer users programming language learning opportunities, but also to provide them with "scaffolding" (after Bruner, 1975) for actually employing SK8Script in their design activity. Expressions that appear in the Language window's Interaction pane as the result of a disclosure are editable and executable as if the user had typed the text into the pane for themselves. All disclosures which appear in the Interaction pane, with the exception of possibility disclosures, are also added to the history list at the bottom of the Language window. By making past and present disclosures immediately accessible and editable, Chart 'n' Art seeks to encourage users to experiment with its programming language. This does not mean that users need to abandon their familiar mouse-based interaction style. Rather, users are free to gradually weave programming expressions into their design strategies. As demonstrated by the analysis of the case studies in Chapter 2, the effective use of programmable applications often requires a balance of language-based and mouse-based interaction.

Experimentation with programming in Chart 'n' Art can initially involve simply changing a value in a disclosed expression and then evaluating it. For example, after Alice drags the left edge of the bar depicted in Figure 4-5b, the disclosure "`set the width of Rectangle 30 in DrawWindow to 6.0`" would appear in the Language window (the actual width value would depend on the final state of the rectangle upon release of the mouse). Even without completely understanding the expression, she could then edit it in the Interaction pane and replace 6.0 with a different value. Pressing the Execute button would confirm that the expression indeed changed the width of the bar. Alice might then make a mental note of her discovery and continue on with her direct manipulation of the chart. Or she might try further editing in the Interaction pane, say by changing the rectangle number 30 to 29 or even changing the property name from `width` to `height`. Alice could use these kinds of experiments to verify hypotheses developed from watching disclosures or to try out variations she had never seen before.

Once users become comfortable with some statements, they can begin to employ the expressions to perform certain design subtasks more accurately and (possibly) more quickly. To begin using expressions does not necessarily mean to have memorized their syntax or vocabulary. A common strategy adopted by some Chart 'n' Art users is to first perform a desired action inaccurately with the mouse for the sole purpose of triggering a disclosure which generates a particular expression in the Interaction pane. This expression acts as a template which they then edit to achieve the desired accuracy.

Possibility disclosures facilitate a different strategy for experimenting with Chart 'n' Art's language. Recall that this disclosure mechanism causes various possible expressions to appear in the Language window's Interaction pane after one or more shapes are selected. Just like alternatives disclosures these expressions can be edited and executed, singly or in combination. For example, when Alice selected the first bar in her chart in Section 4.3.4, a number of `set` expressions appeared in the Language window. She could have edited and evaluated, say, the expression involving the 'bottom' property so that the bottom of the bar bottom was moved to the low temperature for the new month she was plotting. This example illustrates how possibility disclosures can become a means for generating expression templates for more obscure statements that would rarely, if ever, appear in alternatives disclosures resulting from direct manipulation. For users who are wondering if an appropriate expression exists for what they need to do, possibility disclosures not only show them their options, but give users the opportunity to try them out immediately.

5. Chart ‘n’ Art in Perspective

The previous chapter described the basic features of Chart ‘n’ Art and the disclosure mechanism that can help users learn its programming language. To provide a more complete picture, this chapter looks at Chart ‘n’ Art from a number of different perspectives including how it evolved in response to formative feedback, its implementation, its relationship to the dimensions of educational disclosures, and, finally, how it compares to existing systems.

5.1 Formative Feedback and the Evolution of Chart ‘n’ Art

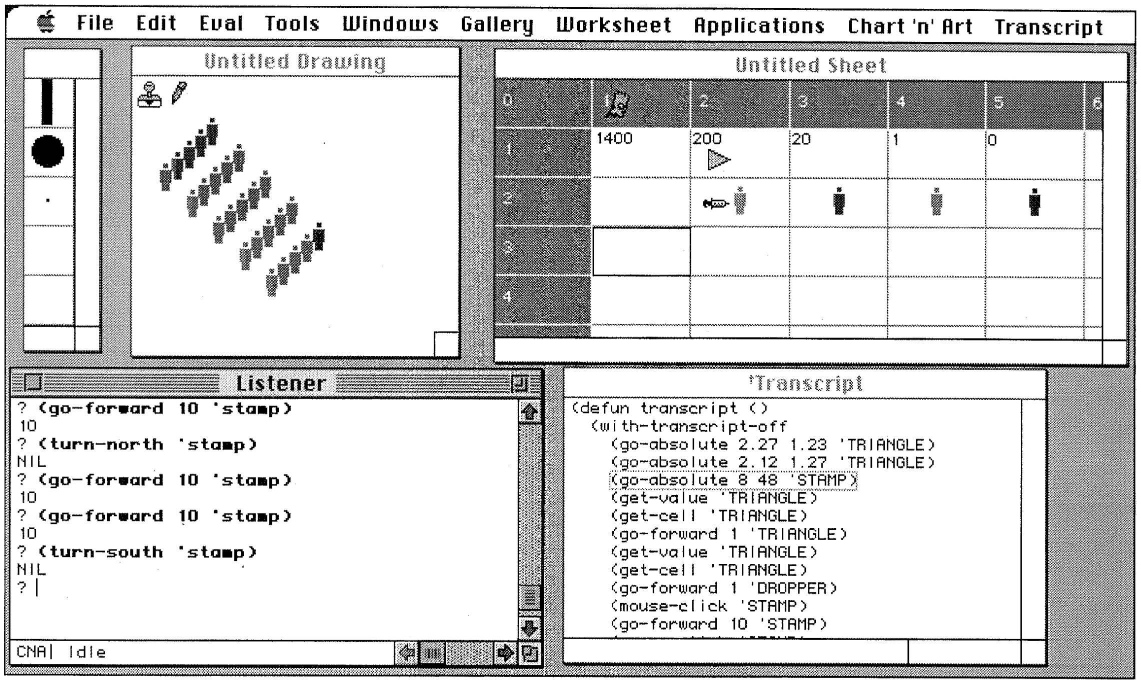
Feedback from informal evaluations by colleagues, a field study described in Chapter 2, and a feasibility study and preliminary evaluation to be discussed in Chapter 6 influenced significant changes in Chart ‘n’ Art over the two-and-a-half years of its development. The earliest version of Chart ‘n’ Art, pictured in Figure 5-1, disclosed only Lisp, not SK8Script, and featured programmable turtles and a spreadsheet for storing both data and chart elements. This section describes Chart ‘n’ Art from an evolutionary perspective by outlining some formative incidents leading from this early “Lisp-based” prototype to the current “SK8-based” implementation described in the previous chapter and used in the summative assessment explained in Chapter 6.

5.1.1 Designers resisted spreadsheets

Chart ‘n’ Art was originally designed around a spreadsheet metaphor with cells able to contain not only textual and numeric data, but charts and chart elements such as data point markers (see Figure 5-1). Even Chart ‘n’ Art programs could reside in individual cells with nearby cells acting as local variables. One of the chief advantages of this unifying spreadsheet theme was the fact that a single spreadsheet file could contain everything needed to recreate a chart using a short program that acted on collections of cells. This metaphor was called into question, however, after users complained the interface was too unusual. Most damning was the fact that designers in the CU publications department field study showed resistance to using spreadsheets in the first place, even for storing raw data. It became clear that much more research was needed to make the spreadsheet metaphor easy to use, and the decision was made to focus instead on improving Chart ‘n’ Art’s disclosure mechanisms within a more familiar drawing environment.

5.1.2 Choice of syntax could interfere with learning

Evaluation of the original Lisp-based version of Chart ‘n’ Art identified a number of common syntactic problems users had with the system’s end-user language. For example, the feasibility study revealed that users often forgot to quote the names of the programmable turtles used for generating charts and extracting values from the spreadsheet. Another problem was with the powerful mapping expression `set-multiple`, which, although simplified from the standard Lisp `mapcar` expression, still proved confusing for subjects in the preliminary evaluation. These same users also had trouble matching parentheses when editing longer expressions within the small interpreter window provided for them in the experiment. Although one could argue that there are long term advantages to users learning the strange but relatively uniform Lisp syntax, an overriding concern was Chart ‘n’ Art’s ability to demonstrate measurable learning effects in a *short term* experiment. What was needed was a more readable end-


File Edit Eval Tools Windows Gallery Worksheet Applications Chart 'n' Art Transcript
Untitled Drawing
Untitled Sheet
1400
200
20
1
0
Listener
CNA| Idle
Transcript

```
? (go-forward 10 'stamp)
10
? (turn-north 'stamp)
NIL
? (go-forward 10 'stamp)
10
? (go-forward 10 'stamp)
10
? (turn-south 'stamp)
NIL
? |
```

```
(defun transcript ()
  (with-transcript-off
    (go-absolute 2.27 1.23 'TRIANGLE)
    (go-absolute 2.12 1.27 'TRIANGLE)
    (go-absolute 8 48 'STAMP)
    (get-value 'TRIANGLE)
    (get-cell 'TRIANGLE)
    (go-forward 1 'TRIANGLE)
    (get-value 'TRIANGLE)
    (get-cell 'TRIANGLE)
    (go-forward 1 'DROPPER)
    (mouse-click 'STAMP)
    (go-forward 10 'STAMP)
```

Figure 5-1. An early version of Chart 'n' Art with programmable turtles (stamp, pencil, turtle, Bart Simpson™) and a spreadsheet (upper left) for storing both data and chart elements. Turtles in the spreadsheet could traverse data sets and extract values while turtles in drawing window could place chart elements.

user language with limited special characters, and, preferably, a more English-like syntax. This was a major impetus for reimplementing Chart 'n' Art in SK8, since the more readable SK8Script language could be used in place of Lisp.

5.1.3 Lack of robustness could hinder formal evaluations

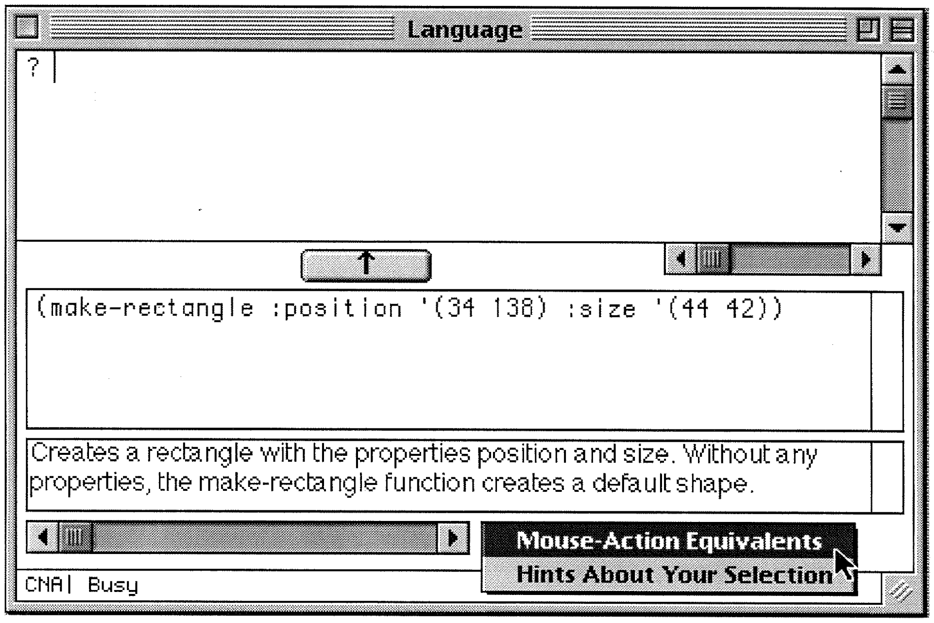
The Lisp-based Chart 'n' Art system tested in the preliminary evaluation had serious reliability problems which, even with constant experimenter vigilance, resulted in frequent interruptions and an occasional reboot. Clearly, this would be problematic for any system being evaluated, but it was especially troubling for Chart 'n' Art, given that its disclosure mechanisms were supposed to support learning-while-doing in a way that was relatively unintrusive. Part of the solution was to limit features in the reimplementation to those that would be critical for the summative assessment. For instance, the ability to save charts and graphs was put off until after the experiment, even though this had been available in the Lisp-based version. This, combined with rigorous pilot testing, enabled Chart 'n' Art to perform almost flawlessly for the thirty subjects in the summative evaluation, never requiring a reboot.

5.1.4 Designers wanted tight control over their tools

The SK8-based version of Chart 'n' Art originally provided on-screen forms for specifying standard chart types such as bar charts and x-y plots. Linked to each of these forms was a code view that automatically updated with every change to the form, new or modified expressions appearing in red. However, observers pointed out that this type of monitoring disclosure offered a very different kind of language learning experience that would require its own testing methodology. Perhaps more importantly, the chart creation mechanisms underlying these forms also violated the clear need for low-level design control demanded by the field study informants. Recall that the professional graphic designers who were interviewed preferred to draw chart elements using the primitive shape drawing tools in packages such as Adobe Illustrator rather than have parts generated automatically by a charting-type program. In the end, the chart forms were left out of the version of Chart ‘n’ Art used in the summative assessment.

5.1.5 Multiple disclosure modes were confusing and effortful

Subjects in the preliminary evaluation and casual users of the Lisp-based Chart ‘n’ Art sometimes missed seeing important disclosures such as `set-multiple` because they were in the wrong disclosure “mode.” This earlier version of Chart ‘n’ Art provided either alternatives disclosures or possibility disclosures, but not both. Users had to select between these two feedback modes with a pop-up menu in the Language window shown in Figure 5-2. Users in the preliminary evaluation seemed either reluctant or forgetful about switching back and forth between the two modes. The fact that users needed to take such an active role in the learning process, i.e. by changing the pop-up menu setting, was also inconsistent with the overall learning-while-doing philosophy behind Chart ‘n’ Art. Finally, as is typical with most moded systems, users had trouble remembering which mode was currently active as was evidenced by subjects in the preliminary evaluation who expressed surprise at seeing certain disclosures only to realize later that they were in the wrong disclosure mode. In response to the above problems, the two disclosure modes were merged into one for the SK8-based version of Chart ‘n’ Art. By default, users would see “Mouse Action Equivalents,” but selection of objects with the mouse would automatically trigger a “Hints About Your Selection” disclosure.



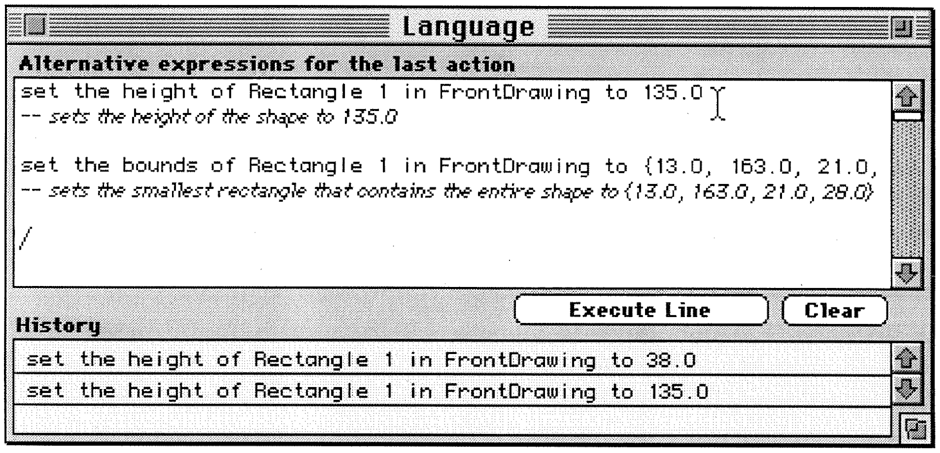
```
(make-rectangle :position '(34 138) :size '(44 42))
```

Creates a rectangle with the properties position and size. Without any properties, the make-rectangle function creates a default shape.

Figure 5-2. The Language window of the Lisp-based Chart ‘n’ Art system. Disclosures appeared in the middle text pane of the window with an explanation for each disclosure in the bottom pane. The pop-up menu beneath the explanations pane selected between two different disclosure modes. The horizontal “Alternatives” scroll bar to the left of the pop-up menu allowed users to browse alternative disclosures. Users could only enter new expressions in the upper Interpreter pane.

5.1.6 Alternatives scroll bar was used only infrequently

Another reason that users were not exposed to certain disclosures was that the Lisp-based Chart 'n' Art prototype could only show one of possibly many alternative disclosures at a time. To see other alternatives, users had to adjust the horizontal scroll bar in the Language window, pictured in Figure 5-2. Without prompting, however, subjects in the preliminary evaluation were rarely motivated to manipulate the scroll bar in order to view other alternatives. In addition, the scroll bar itself did not immediately suggest to subjects in the preliminary evaluation that it could be used to switch to different alternatives. Instead, many in the preliminary evaluation thought the scroll bar would simply adjust their view of the disclosure shown in the middle text pane. The solution adopted in the SK8-based Chart 'n' Art system was to output all alternatives and their explanations to a single vertically scrollable text pane as shown in Figure 5-3. Users could choose to make the pane any size they wanted, depending on how many alternatives they were interested in seeing at one time. The semantics of the vertical scroll bar, then, were much clearer: adjusting it would allow one to view the alternatives listed above or below the current window of disclosures.



```
set the height of Rectangle 1 in FrontDrawing to 135.0
-- sets the height of the shape to 135.0

set the bounds of Rectangle 1 in FrontDrawing to {13.0, 163.0, 21.0,
-- sets the smallest rectangle that contains the entire shape to {13.0, 163.0, 21.0, 28.0}
```

```
set the height of Rectangle 1 in FrontDrawing to 38.0
set the height of Rectangle 1 in FrontDrawing to 135.0
```

Figure 5-3. The simpler and more compact Language window used in the SK8-based Chart 'n' Art system. As many alternatives as will fit (and their explanations as comments) are listed in the scrollable Interaction pane at the top of the window. At the bottom is the History which replaces the Transcript window in the Lisp-based version of Chart 'n' Art.

5.1.7 Disclosures could clutter designers' screens

Disclosure presentations in the Lisp-based Chart 'n' Art system occupied a variety of text panes spanning two separate windows: the Language window in Figure 5-2 and a scrolling Transcript window which maintained a complete list of all disclosures shown in the current session. Although three out of the five subjects in the preliminary evaluation felt the disclosure windows would not get in the way of their work, one commented that it might become problematic with screens smaller than 17 inches diagonal. Indeed, informal discussions with potential users revealed concerns that Chart 'n' Art's disclosure mechanisms might clutter limited screen space. Clearly, a more compact presentation of disclosures would allow users to better focus their attention on the information they needed to learn. It also would also mean that disclosures could be less distracting for users trying to get their work done. The Language window in the SK8-based Chart 'n' Art shown in Figure 5-3 with its Interaction pane and History pane was designed to combine the functionality of the Transcript window and previous Language window into a single compact view.

5.1.8 Mechanisms for reusing disclosed expressions were awkward

One subject in the preliminary evaluation complained that reusing disclosures required an "awful lot of button clicks." He was right. Reusing expressions in the Language window for the Lisp-based Chart 'n' Art (see Figure 5-2) meant manipulating the alternatives scroll bar to find the desired disclosure, then pressing the up arrow button in the middle of the window to introduce a disclosure into the Interpreter pane at the top of the window. An alternative approach involved copying and pasting text between the Transcript window and the Interpreter pane. Either solution was cumbersome and clearly did not encourage designers to explore the end-user language being disclosed. The solution was the multipurpose Interaction pane found in the current SK8-based version (see Figure 5-3) which enables users to edit and execute the original disclosures directly without having to copy the text into a separate view.

5.1.9 Users were unsure how to refer to drawing objects

A common misconception by many users of the early versions of Chart 'n' Art was that programming language expressions for managing shapes, such as for coloring or moving objects, would automatically operate on the current selection. This was not an unreasonable assumption, given that disclosures usually only appeared after a user selected an object and performed some direct manipulation action. The object was usually still selected after the disclosures appeared, so these expressions could, in theory, reexecute properly if they did indeed use the current selection as an argument.[9] To make matters worse, this assumption apparently led some users to interpret explicit object references such as `Rectangle-1` in the disclosure "`(set-height Rectangle-1 20)`" as a parameter meaning "the currently selected rectangle." The confusion over drawing object reference inspired the development of the selection labeling disclosure technique used in the SK8-based Chart 'n' Art. Selection labels were designed to increase users' awareness of the SK8Script for explicitly referring to drawing objects, a very useful class of expressions. The correspondence between these selection labels and any disclosed expressions involving the labeled object highlighted to users that shape management expressions did not necessarily operate on the current selection.

5.1.10 Users did not necessarily attend to disclosures

Of considerable concern was the fact that some users did not necessarily attend to expressions appearing in the disclosure window(s) in the Lisp-based and early versions of the SK8-based Chart 'n' Art. In the feasibility study, for instance, when asked to type a particular command into the interpreter window, one subject resorted to trial and error even though an example of the command was in plain view, but in a separate disclosure window. It was also not encouraging that one of the informants in the CU Publications field apparently had not noticed the simple disclosures appearing in the margin of her active Adobe Illustrator window. As shown in Figure 5-4, Illustrator displays the name of the current tool being applied to a drawing in the bottom left margin of the window. Even when in the active window, it seemed users would not

---

[9] This is, in fact, characteristic of the expressions many "recordable applications"(see Section 5.2) such as the MacOS Finder disclose. For example, when the author selected the file for this chapter and chose Duplicate from the Finder menu, the MacOS Script Editor recorded the following AppleScript program:

```
tell application "Finder"
    select file "Chart 'n' Art in Perspective" of startup disk
    duplicate selection
end tell
```

necessarily attend to disclosures. It was feared that the alternatives and possibility disclosures in a separate Language window might by themselves be too subtle to have measurable educational value in a short term study. These concerns were a major impetus for the addition of mouse disclosures and to a lesser extent, selection labeling. The goal was to ensure exposure to programming language expressions through disclosures appearing not just in the active window, but by the mouse pointer within the user's current field of view. Clearly, these "in-your-face" tactics are not always desirable, and should ultimately be under user control in any commercial self-disclosing system.

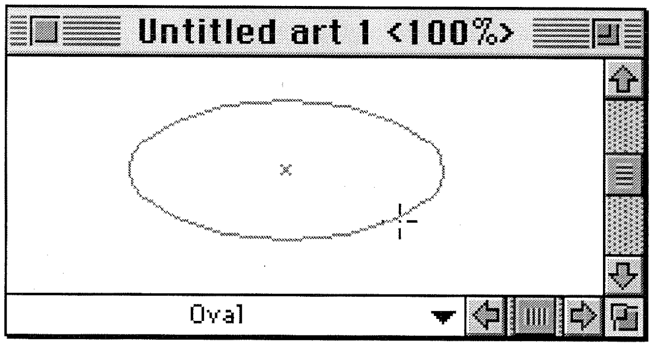


Figure 5-4. Adobe Illustrator's tool name disclosure appearing in the bottom left corner of the drawing window.

5.1.11 Users were confused about the meanings of possibility disclosures

Initial pilot tests of the SK8-based Chart 'n' Art prior to the summative assessment revealed fairly consistent confusion over the semantics of possibility disclosures. As they do in the current version of Chart 'n' Art, possibility disclosures during the initial pilot tests listed property setting expressions relating to the active selection. However, unlike the current version, the values being assigned to each property represented random perturbations of the current state. Evaluating any of these possible expressions would then cause the selected shapes to change slightly, say from yellow to purple, or from 100 units tall to 90 units tall. The reasoning behind this approach was that by executing possibility disclosures without modification users could immediately see how certain expressions could help them alter their drawing. But pilots indicated that they expected the values to represent the actual state of the drawing, not its potential state. Some wanted to use possibility disclosures to verify that they had adjusted a shape properly by direct manipulation. For example, after widening a shape with the mouse, they wanted to be able to then select the shape, find the width-setting expression in the possibility disclosure, and check the value against the intended width. Rather than try and educate users about the unintuitive semantics of possibility disclosures, possibility disclosures were altered to match users' expectations. The current version of Chart 'n' Art now discloses expressions whose values represent the current state of the selection. Evaluating these expressions without modification has no effect on the drawing.

5.2 Chart 'n' Art Implementation

As already noted, Chart 'n' Art has undergone substantial revision from an early Lisp-based version developed in Macintosh Common Lisp to a recent SK8-based reimplementation developed with the prototype SK8 authoring environment from Apple

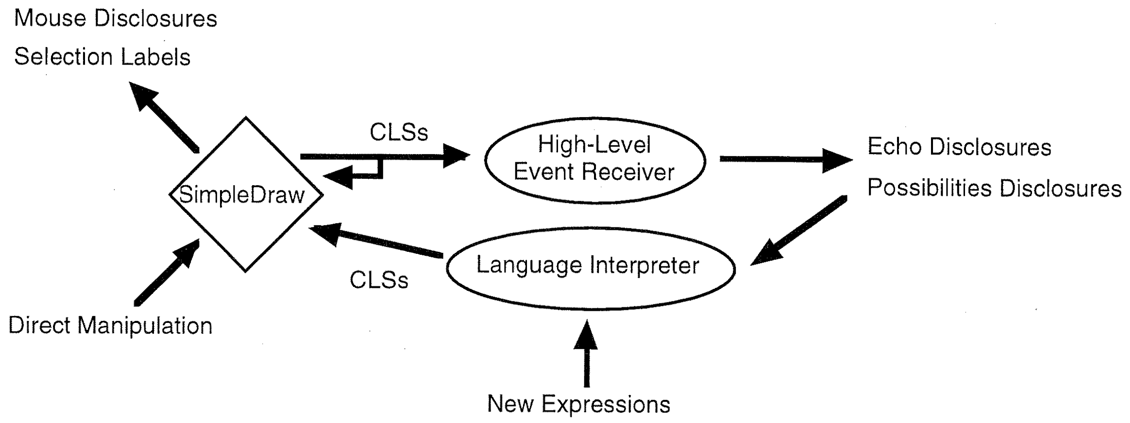


Figure 5-5. The three main components of the SK8-based Chart 'n' Art architecture.

Computer, Inc. Both run on Mac OS computers. This implementation discussion focuses on the current SK8-based version.

The Chart 'n' Art architecture consists of three main components as illustrated in Figure 5-5: a drawing tool called SimpleDraw, a high-level event receiver, and a language interpreter. SimpleDraw maintains the drawing windows, palettes and most menu items, and is responsible for generating mouse disclosures and selection labels in response to low-level direct manipulation events. The high-level event receiver monitors interaction with SimpleDraw and presents alternatives and possibility disclosures when appropriate. The language interpreter allows users to modify disclosed commands or type new ones and then execute them in SimpleDraw. These commands are sent to SimpleDraw in the form of Canonical Language Structures (CLSs), abstract high-level events designed especially for self-disclosing systems. Canonical Language Structures are also used to transmit information from SimpleDraw to the high-level event receiver. When SimpleDraw detects the completion of a direct manipulation action it broadcasts a CLS to any interested receivers and sends itself a copy of the CLS so that the command specified by the user's gesture is carried out.

The above data-flow model is similar in many ways to that of a "recordable application" as defined by Apple Computer, Inc. in its Open Scripting Architecture (OSA) (Apple Computer Inc., 1995) for Mac OS. In fact, CLSs are supersets of Mac OS's high-level events, Apple Events. This means that SimpleDraw can be recorded and controlled by Apple's Script Editor and other Apple Event compatible tools such as UserLand Frontier (UserLand Software Inc., 1996). Like an Apple Event, a CLS is an application independent, programming language neutral representation of an operation which can be converted later into an actual expression. Chart 'n' Art makes use of a separate disclosure service to transform CLSs into disclosed expressions as shown in Figure 5-6. This service consults a language definition database based on the Apple Event Terminology resource standard, also part of OSA.

The OSA model was chosen as a foundation for Chart 'n' Art, since OSA's modular architecture could, in theory, allow disclosing to take place independently of the design tool.[10] In practice, the OSA model needed a number of extensions to support

[10] An inspiration for this approach is Eager (Cypher, 1993) which uses OSA to enable recordable applications to be programmed by demonstration with minimal

application independent self-disclosure—some of which were done for Chart 'n' Art, others were left for future work to be described in the concluding chapter. Canonical Language Structures represent one of the OSA extensions that was implemented. They have the following advantages over standard OSA Apple Events:

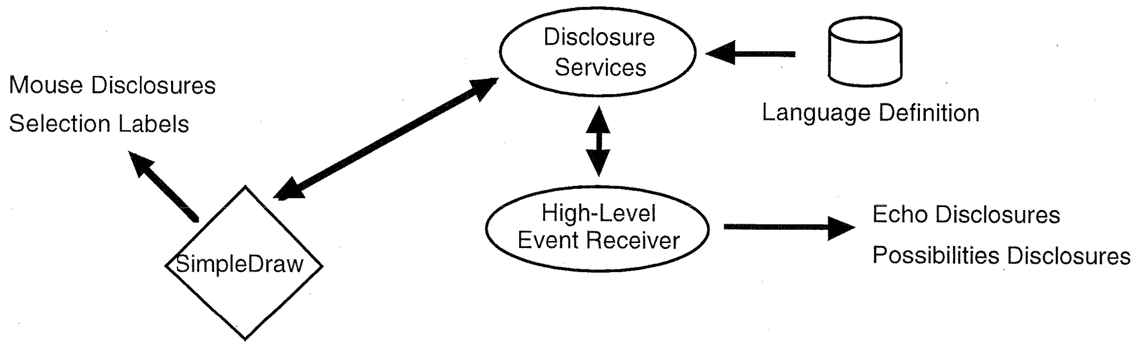


Figure 5-6. The role of the disclosure services component in helping the drawing tool, SimpleDraw, and the high-level event receiver present feedback in a variety of programming languages.

- CLSs provide an additional "alternatives" parameter for listing equivalent expressions in the case of alternatives disclosures and possibilities in the case of possibility disclosures.

- CLSs can specify language structures not just events. Apple Events are limited to describing procedure invocation, object specification, and the getting and setting of object properties. CLSs can specify variable assignments, looping constructs, and numeric expressions to name a few possibilities.

- CLSs can be arbitrarily nested. This allows a CLS to specify a complex expression such as
  `set dup to copy Rectangle 30 in DrawWindow`
  where a copy operation is embedded within a variable assignment.

- CLSs provide an additional "explanation" parameter for providing a custom explanation for an expression.

Using the more expressive CLS, Chart 'n' Art's SimpleDraw and high-level event receiver components can disclose more interesting and more educational programming language statements than would otherwise be possible using Apple Events and yet still remain application and language independent. These components send a CLS to the disclosure services module which uses the language definition database to translate the CLS and its alternatives into the target programming language and automatically generate explanatory text for each CLS unless it has a custom explanation. The disclosure service then returns text strings for each expression and for each explanation for the components to display either by the mouse, in the selection label, or in the Language window.

---

modification. Similar in spirit is the Application Independent Demonstration Environment (Piernot & Yvon, 1993) which provides an OSA-like architecture for programming by demonstration systems under the Smalltalk environment.

Although CLSs allow disclosures to be specified in a language independent fashion and explanations can be generated automatically by disclosure services, responsibility currently still rests with the SimpleDraw module to actually create the appropriate CLSs. Not only must SimpleDraw construct the CLS to send to itself to carry out a direct manipulation command (the definition of a "factored" application in OSA), but it also must generate any alternative CLSs that should be disclosed along with the primary CLS. Clearly this is an unreasonable burden for developers of anything but a prototype system such as Chart 'n' Art. The concluding chapter describes some techniques envisioned for minimizing the modifications needed to render tools self-disclosing.

5.3 Dimensions of Chart 'n' Art Disclosures

Another perspective on Chart 'n' Art is to analyze its self-disclosure mechanisms in terms of the dimensions of educational disclosures described in Chapter 3. Figure 5-7 uses these dimensions to summarize how Chart 'n' Art matches up to the suggested profile of an educationally effective self-disclosing system. Let us discuss each dimension in the figure one at a time.

5.3.1 Adaptivity

The current SK8-based version of Chart 'n' Art does not adapt its disclosures to suit the needs of each user individually. Identical direct manipulation actions trigger identical disclosures, regardless of the user's programming background and regardless of any prior experience using Chart 'n' Art. The Lisp-based Chart 'n' Art was somewhat adaptive in that the system kept count of the various types of disclosures shown to each user and employed these statistics to determine which of the alternative expressions to show first. All alternatives were ordered in increasing difficulty as best as possible using a one-dimensional scale. After a user had seen one alternative a certain number of times, the system assumed the user understood this expression, and automatically began showing the next most difficult alternative first. A similar "light-weight user model" could easily be implemented for the current version of Chart 'n' Art. However, the utility of such a simplistic approach is questionable. More sophisticated user modeling was considered beyond the scope of this research and is already a well-researched topic in educational computing, especially for the teaching of programming languages (see the discussion of the Lisp tutor, Smalltalk scaffolding, Lisp-Critic, and COACH in Section 5.4.4).

5.3.2 Explorability

The ease with which users can experiment with Chart 'n' Art disclosures gives the system a high degree of explorability. Most disclosed expressions appear in the Language window's Interaction pane where they can be immediately edited and evaluated. Previous disclosures can be recalled from the history list and sent to the Interaction pane with a single click of the mouse. Explanatory text accompanies alternatives- and possibility disclosures to help users make sense of the expressions. Undoubtedly the most useful addition to these facilities would be the ability to undo the effects of the most recently evaluated programming expression as is possible with some programming by demonstration systems such as AIDE (Piernot & Yvon, 1993). Also useful would be hypertext links to related expressions and more in-depth explanations.

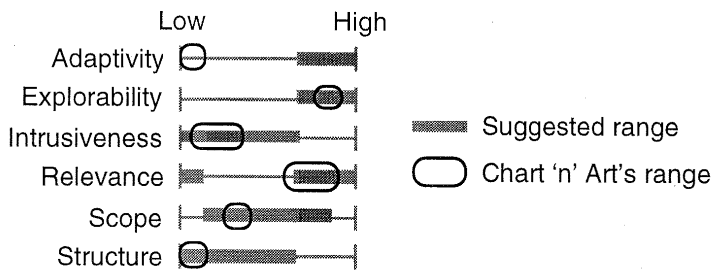


Figure 5-7. Chart 'n' Art's match to the suggested profile of an educationally effective self-disclosing system.

5.3.3 Intrusiveness

Chart 'n' Art's four main disclosure mechanisms each have a different degree of intrusiveness. Mouse disclosures are probably the most intrusive in that expressions are presented at the cursor, directly in the user's field of view. Next are selection labels which, although not necessarily in the user's field of view, can clutter the drawing area. In some ways, these labels are more troublesome than mouse disclosures in that they persist as long as a selection is active. Users who are annoyed by this, however, can always deselect and the most recent selection label will disappear. Less intrusive are possibility disclosures which appear in the Language window, completely separate from any drawings. This window can be made any size and positioned in the least obstructive part of the screen to minimize distractions. A current drawback of possibility disclosures is that because of their computational overhead, they slightly reduce the system's responsiveness to mouse actions made directly after selecting one or more shapes. This could be improved by making disclosure printing interruptable. Finally, the least intrusive disclosure mechanism is alternatives disclosures which use the same separate Language window, but do not incur the performance penalties of possibility disclosures.

Overall, Chart 'n' Art's disclosures are relatively unintrusive, especially when compared to systems which require explicit user initiative (e.g. users must request to see disclosures) or acknowledgment (e.g. users must dismiss informational dialog boxes). Ideally, users should be able to selectively enable/disable Chart 'n' Art's disclosure mechanisms via preferences dialogs, perhaps similar to the intrusiveness slider used in (Nakakoji et al., 1994).

5.3.4 Relevance

Disclosures in Chart 'n' Art are implicitly relevant to user activity because the programming expression feedback is in direct response to interaction with the system. Probably most relevant are mouse disclosures since they continuously display equivalent expressions to what the user is *currently* doing by direct manipulation. Next are alternatives disclosures and selection labels since they present expressions corresponding to the most recent operation or selection. Least relevant are possibility disclosures, since their output is a sequence of expressions for acting on an essentially random list of object properties.

The above characterization of Chart 'n' Art's self-disclosure mechanisms measures relevance by temporal proximity to associated user interaction. This is reasonable, if the goal is simply that users learn from disclosures, since temporal proximity has been shown to be important for learning from examples and analogical reasoning (Lewis, 1988; Lewis, Hair & Schoenberg, 1989). Another possible relevance measure is how close the disclosed expression are to the kinds of statements that would complement rather than replace direct manipulation operations. Using this measure, possibility disclosures are more relevant than the other Chart 'n' Art mechanisms, since they present language-based operations that users could *potentially* apply to the active selection, but that might not be achievable with the mouse.

5.3.5 *Scope*

Chart 'n' Art disclosures mainly cover a limited, but useful set of SK8Script expressions: shape properties, selection expressions, assignment statements, mapping, and iteration. These expressions were identified by analyzing real-life design tasks such as those described in Chapter 2. A more rigorous "programming walkthrough" (Bell et al., 1994) of a variation of the temperature chart task at *The Daily Camera* helped further guide the choice of disclosed expressions. This analysis is discussed further in Chapter 6. In addition to disclosing a number of useful programming techniques, Chart 'n' Art also attempts to familiarize users with a variety of ways for composing expressions. Unlike simple macro recording which shows only one-line procedure calls (as in Apple's Script Editor), Chart 'n' Art's alternatives disclosures reveal sequenced statements, nested expressions, and programming plans involving temporary variables. Also unique to alternatives disclosures is the display of multiple equivalent expressions which further increases the scope of programming language expressions seen by users.

5.3.6 *Structure*

Disclosures in Chart 'n' Art currently have little, if any, structure. That is, there is no explicit attempt to sequence programming language expressions in a particular order. Disclosed expressions and—in the case of alternatives and possibility disclosures—their accompanying explanatory text cannot assume that any other expressions had been seen previously. One should note that certain implicit patterns in the use of Chart 'n' Art's drawing tools does lead to a number of short predictable sequences of disclosures. For instance, only after creating a model shape do users duplicate it; and only after creating a number of shapes, do users typically distribute them horizontally, vertically, or diagonally. Disclosures for duplicating and distributing are indeed more complicated and less frequent than those for creating, position, and sizing shapes, so one could argue that there is accidental structure to Chart 'n' Art disclosure mechanisms. On the other hand, it is questionable whether the basic disclosures that novice users see while creating and adjusting individual shapes prepare them sufficiently for the more advanced expressions that they encounter shortly afterwards when attempting to duplicate or distribute those shapes. The time frame for these "accidental" sequences may simply be too rushed. The philosophy behind Chart 'n' Art is that the most important sequence for disclosures to follow is the sequence of user interaction with the system, not a rigid programming language curriculum. Responsibility is left with the user to link together the appropriate language learning opportunities afforded by disclosures into a personally meaningful, long term learning experience.

5.4 Related Systems

This final perspective on Chart 'n' Art examines related systems in the fields of programmable applications, programming by demonstration, and intelligent tutoring systems.

5.4.1 Programmable Applications for Information Graphics

Programmable applications are becoming increasing prevalent as software tools become more complex and user communities mature. For instance, as mentioned in the introductory chapter, Microsoft has now integrated its Visual BASIC language into the most recent versions of its personal productivity software including Word and Excel. Early on, this report described AutoCAD, one of the pioneering programmable applications with some rudimentary self-disclosing features. Other commercial programmable applications that have long supported end-user programming include Director (Macromedia Inc., 1996a) and Mathematica (Wolfram Research Inc., 1996). With Chart 'n' Art as a comparison, let us now consider two programmable applications in particular, SchemeChart and Excel, which also support information graphics.

SchemeChart. SchemeChart (Eisenberg & Fischer, 1994) was a predecessor to Chart 'n' Art which included a direct manipulation interface for editing charts by hand and an integrated Scheme interpreter for customizing built-in graphing functions or creating entirely new kinds of charts. Unlike Chart 'n' Art, SchemeChart's direct manipulation operations primarily supported painting at the pixel level; no tools were provided for creating or manipulating primitive shapes such as rectangles and ovals. On the other hand, SchemeChart offered a richer language than Chart 'n' Art with graphing-specific expressions for creating components of charts including points and axes and methods for composing these elements into a complete illustration.

SchemeChart had some basic facilities for familiarizing the user with this language. For example, in the spirit of the Explainer system (Redmiles, 1993), users could select a sample chart from a palette and view an annotated code listing of the Scheme function used to create the sample. In contrast, Chart 'n' Art explores the potential for users to learn incrementally from shorter examples relating to direct manipulation activity. SchemeChart users could also enter a "query mode" in which the selection of a chart element caused the system to disclose a list of related procedures for operating on that element. This technique was the precursor to Chart 'n' Art's possibility disclosures.

Visual Basic and Excel. The Microsoft Excel 5.0 spreadsheet now integrates a version of Visual Basic which allows users to automate everyday tasks, add custom features, and even create complete applications. Visual Basic can not only be used to organize and format data sets, but also to specify drawing objects such as rectangles, ovals, and lines. Users can create and manage these same objects by directly manipulating drawing tools available in a special Excel toolbar. In macro record mode, users can set up Excel so that while they employ the mouse to manipulate shapes, equivalent Visual Basic commands are echoed to a separate window. Although this feedback is similar to alternatives disclosures in Chart 'n' Art, Excel's macro recording facilities are not specifically intended to support end-user programming learning. Indeed, Excel's *Visual Basic User Guide* claims "...you don't have to be a programmer to create macros; in fact, you don't even have to learn Visual Basic to put macros to work" (Microsoft Corp., 1995). Instead, macro recording is provided primarily as a means for specifying simple sequences of operations without typing or even looking at Visual Basic. Only through the clever use of Excel's viewing commands can users set up the dual window display necessary for seeing macros as they are being recorded—a trick not mentioned in the manual.

Even when users can view Excel's disclosures, their learning opportunities are limited. With few exceptions, the scope of macros generated during recording encompasses only single-line procedure calls operating on the current selection. Exploration is limited to editing the macro and executing it from the start. Users cannot evaluate individual expressions that appear in the macro being recorded nor does Excel attempt to annotate recorded expressions with explanatory text. An advantage Excel has over Chart 'n' Art is its high-level charting functions for creating standard graph types such as column and line charts. These functions do appear in macro recordings; however, users cannot inspect or customize them as was possible in SchemeChart.

5.4.2 Programming Graphics By Demonstration

A number of systems allow users to create custom charts through a different type of programming known as "programming by demonstration" or "programming by example." With this approach users indicate their intentions by presenting examples of partial or complete charting solutions to the system. The system is often able to infer from these examples a limited set of declarative constraints or procedures which simplify the production of similar graphs.

Sage. The Sage system (Roth, Kolojejchick, Mattis & Goldstein, 1994) is comprised of three main components: SageBrush, for assembling graphics from primitive objects, SageBook, for browsing previously created graphics, and SAGE, a knowledge-based presentation system that automatically designs graphics and also interprets users' specifications. In SageBrush users employ direct manipulation to identify the graph elements for a particular design and to indicate how the properties of these graph elements map to their data. The resulting "sketch" is sent to SAGE where a final illustration is generated. Both sketch and final output can be stored in SageBook for later reuse.

The sketch created in SageBrush is actually an abstraction of the desired chart typically consisting of an "encoder" defining a spatial organization (e.g. network, table, grid), a single instance of each graph element (e.g. mark, bar, line, text box) called a "grapheme," and symbols indicating connections to the data. For example, a simple line chart would consist of a 2D spatial coordinate display encoder and a single arbitrarily positioned line representing all of the line segments in the chart. By dragging cells from Sage's built-in spreadsheet onto a graphical view of the properties of the representative line, a user could indicate how to map the data to the line chart.

In essence, SageBrush's sketch is the "program" which declaratively specifies a design. There is no need for a second language-based view on user actions, because programming and drawing are one in the same activity. On the other hand, it is questionable how approachable designers would find this type of drawing since it is not What You See Is What You Get (WYSIWYG). The reliance on spreadsheet skills is also a concern in light of the hesitation designers in the case studies in Chapter 2 showed toward such tools. Finally, Sage's graphical language of encoders and graphemes lacks the ability to perform numerical calculations useful for, say, interpolating points in a line chart to create a smooth curve; and the ability to define imperative procedures for performing animation or generating fractal patterns.

Gold. Graphs and Output Laid-out by Demonstration (Gold) (Myers, Goldstein & Goldberg, 1994) is similar to Sage in that drawings made by users also serve as specifications for an automatically generated complete chart. Like Sage, Gold also relies critically on users' ability to interact with a built-in spreadsheet. The main difference between the systems is that Gold infers more from the user's drawing than Sage does of a SageBrush sketch. For instance, in creating a column chart, when a user colors a sample bar, Gold assumes this color should be used for the rest of the bars. When the bar is placed on the horizontal axis, Gold infers that any linked data should map to the height of the bar. If the user then selects the bar and then a cell in the spreadsheet, Gold decides that the height of the bar should be determined by data in the selected cell. Finally, when the user draws a second bar Gold infers that a multiple-column chart is intended and automatically generates what it determines to be the remaining bars and maps all the data in the column of the previously selected cell to their heights.

The fact that the final chart is generated in the same space as the user's initial drawing brings Gold somewhat closer to WYSIWYG than Sage, but Gold's inferencing abilities can lead to a number of surprises. For instance, Gold will usually go back and adjust the attributes of the examples drawn by the user, even if the user never intended for them to be "neatened." It is also possible for Gold to chart the wrong data entirely, although Gold does support means for users to directly or indirectly edit the system's assumptions and force it to refine its inferences.

5.4.3 Recordable Programming by Demonstration Systems

Closely related to Chart 'n' Art are recordable programming by demonstration systems which emphasize a dual view of user interaction: a highly graphical view which users operate by direct manipulation, and a more abstract textual or graphical-textual "recording" of a formal program. This combination fits the definition of self-disclosure, since such systems are volunteering information about their programming language. A number of systems fit this description including SmallStar (Halbert, 1993), Pursuit (Modugno, Corbett & Myers, 1995), and to a limited extent GNU Emacs (Stallman, 1981). This section focuses on a few of these systems specifically designed for the creation of complex graphics by demonstration.

Juno-2 and Geometer's Sketchpad. Juno-2 (Heydon & Nelson, 1994) and Geometer's Sketchpad (Jackiw & Finzer, 1993) are constraint-based systems for generating everything from fever charts to fractal patterns. Users typically begin an illustration by creating points and lines via direct manipulation tools. The tools in both systems are more primitive than those in standard drawing packages with no support for creating rectangles or other polygons with a single mouse gesture. Additional points, e.g. midpoints, are specified in relation to existing points and lines. Users can also indicate relations between existing objects, for example to specify that lines should be parallel. Meanwhile, a second window is designed to echo the corresponding textual representation of these objects and constraints using a declarative programming language. The language behind both systems also allows users to name objects and employ imperative constructs such as procedure invocation, enabling iterative and recursive patterns useful for repeating graphical elements and fractals. Unique to Juno-2 is the fact that the graphical and textual views are linked in both directions: users can directly edit the program text which automatically triggers changes in the graphics window.

Like Excel, the linked views in Juno-2 and Geometer's Sketchpad do not seem to be designed with language learning specifically in mind. Since the linked textual view provides a complete specification of the graphics, the scope of these disclosures can be extensive. However, unlike possibility disclosures in Chart 'n' Art, the systems make no attempt to expose users to expressions beyond what they have specified by direct manipulation. Also lacking is the explanatory text that appears in Chart 'n' Art's Language window or linked sample code as in SchemeChart.

Mondrian and Chimera. Mondrian (Lieberman, 1993) and Chimera (Kurlander, 1993) are programming by demonstration systems that record a "comic strip" rather than a textual program. Users of both systems create examples using standard direct manipulation drawing tools such as for producing rectangles and ovals. As an illustration is produced in an editor window, these systems present an enfolding sequence of representative screen snapshots in another window with textual annotations for additional information such as command names. These "comic strip" sequences can be named and parameterized for later reuse. In Mondrian, users must inform the system that they wish to create a command prior to performing the drawing operations that are to be abstracted and "neatened." Chimera comic strips, on the other hand, are generalized after the fact by selecting the frames to be incorporated into a procedure. Although users can employ comic strip programs in Mondrian and Chimera to automate repetitive tasks such as aligning shapes or generating repeated shapes, neither system seems to provide means for mapping data to graphical elements. Since the languages for programming both systems is almost entirely graphical, it is unclear how a user would provide such data without violating the consistency of their program representations.

The developers of Mondrian and Chimera designed the comic strip feedback of both systems to be highly readable and understandable by beginning users. Lieberman's approach includes a consistent "domino" design for both Mondrian's tools and comic strip frames, and the use of voice synthesis to describe the system's inferences in natural language. Kurlander and Feiner's graphical histories for Chimera evolved over several years (as described in Kurlander & Feiner, 1993) to the current form which makes use of zooming, highlighting, and detail suppression to draw users' attention to the most important elements of a comic strip. These techniques for improving the readability of disclosures is lacking from Chart 'n' Art. On the other hand Chart 'n' Art offers additional feedback mechanism such as possibility disclosures, mouse disclosures, and selection labeling that are absent from Mondrian and Chimera.

5.4.4 Learning Environments for Programming

Although all of the graphics tools described so far feature an integrated programming language of one kind or another, none is specifically designed to support the learning of that language. Indeed, it seems that Chart 'n' Art is unique in its emphasis on language learning in the context of design tool use. However, Chart 'n' Art is certainly not the first software to support programming language learning in general. An example of an early environment for programming language learning is the Spade-0 Logo system(Miller, 1982). Below some of the more recent contributions to the field are compared to Chart 'n' Art.

Lisp Tutor. Anderson's Lisp tutor (Anderson et al., 1995) is designed to complement classroom instruction by providing students with a supportive environment for working out a limited number of programming assignments. The system consists of two main windows, one for the user to enter code and the other for the computer-based tutor to present exercises and provide feedback. As the student enters a program, the Lisp tutor checks each symbol to ensure the student's code is following one of several known solution strategies. So long as the student remains on one of these known solution paths, the system simply acts as a structured editor by providing templates for function calls and helping to balance parentheses. However, if the student enters an improper symbol, the system immediately informs the student that there is a problem and gives him or her the opportunity to correct it. Several such errors in a row cause the system to offer the student an explanation for the problem and help in fixing it. The student can also explicitly request these explanations and assistance at any time.

The Lisp tutor obviously represents a very different pedagogical approach from Chart 'n' Art. Students are not free to work on their own problems, but are instead constrained to solving a fixed set of generic programming exercises.[11] On the other hand, by limiting the problem sets, it becomes practical for domain experts (in this case programming instructors) to build powerful production rules into the system that model good solutions to the problems and anticipate common errors. In many respects, the Lisp tutor matches the profile of an educationally effective self-disclosing system by providing feedback that is highly adaptive, that can be further explored by users, and that is highly relevant to the task at hand. However, the Lisp tutor lacks a surrounding environment, such as Chart 'n' Art's direct manipulation drawing facilities, that would allow learners to apply newly learned programming skills in meaningful ways. The responsibility for contextualizing abstract programming knowledge is apparently left to the traditional instruction that the Lisp tutor is designed to accompany.

"Minimalist" Scaffolding for Smalltalk. To help users quickly learn Smalltalk programming, Rosson, Carroll, and Bellamy developed a hands-on curriculum using a customized Smalltalk environment (Rosson, Carroll & Bellamy, 1990). The curriculum consists of a set of example-based learning scenarios grounded in the "minimalist" model of instruction. According to Rosson et al.:

> *Minimalism emphasizes the streamlining of instructional materials, and a task-oriented approach to determining and organizing the content of manuals, tutorials or other educational artifacts. The goal is to support users in accomplishing real and meaningful tasks quickly, and to allow them to take advantage of their existing task knowledge in learning about a new system.*

The authors focused their particular Smalltalk curriculum on the understanding and modification of a blackjack game program. To help users concentrate on the learning activities, two scaffolding mechanisms were incorporated into the Smalltalk environment: a special "Bittitalk" browser which limits classes and methods to only those relevant to the blackjack implementation, and the View Matcher which provides an integrated picture of the running blackjack application, including relevant commentary.

As with the Lisp tutor, it is difficult to compare Rosson *et al.*'s Smalltalk learning environment with Chart 'n' Art, as the systems represent very different educational philosophies. The explicit goal of the Smalltalk curriculum is to enable learners to develop code for an interactive application in four hours. Chart 'n' Art takes a less specific, longer term view of learning as a by-product of the user carrying out everyday tasks. On the other hand, by constraining the learning tasks, Rosson et al.'s Smalltalk curriculum is able to stage the introduction of programming concepts, anticipate problems, and provide detailed "canned" commentary within the scaffolded programming environment.

Lisp-Critic and COACH. Lisp-Critic (Fischer, 1987) and COgnitive Adaptive Computer Help (COACH) (Selker, 1994) are distinguished from the above learning environments in that they are intended to provide educational support for arbitrary programming projects. Users are free to pursue personally meaning problems in the

---

[11] The following is a sample exercise from (Anderson et al., 1995):

Define a function called "create-list" that accepts one argument, which must be a positive integer. This functions returns a list of integers between 1 and the value of the argument, in ascending order.

Lisp language and the systems attempt to offer feedback in the form of suggestions for more readable or more efficient code in the case of Lisp-Critic; and examples, descriptions, or syntax in the case of COACH. Both systems construct a model of a user's programming experience based on analysis of their code and combine this information with a Lisp knowledge base to adapt feedback accordingly. A major difference between the two systems is that while Lisp-Critic requires users to take the initiative and request a critique of their code, COACH is a self-disclosing Lisp programming environment: as users type expressions, the system provides constant feedback in an adjacent window pane to the editor.

Lisp-Critic and COACH share the same incremental learning philosophy embodied in Chart 'n' Art. Unlike Chart 'n' Art, however, these systems provide little help for beginning programmers, since users must be able to write some initial code before Lisp-Critic or COACH can offer help. On the other hand, their ability to adapt to a user's level of expertise means that Lisp-Critic and COACH can better ensure that novice programmers advance their skills.

6. Evaluating Self-Disclosure

An essential component of this report is the evaluation of the self-disclosure approach to teaching end-user programming. This chapter reports on a controlled summative assessment of Chart 'n' Art designed to quantitatively and qualitatively measure the nature of programming learning resulting from disclosures. As background to this experiment, two prior studies are presented which helped assess the feasibility of the self-disclosure approach and evolve the design of the Chart 'n' Art prototype.

6.1 Initial Studies

6.1.1 Feasibility Study

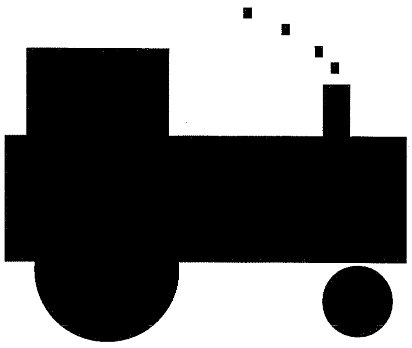
A feasibility study was designed to assess how much users would attend to programming language disclosures and whether exposure to the disclosures would indeed enable users to begin employing programming in a drawing tool. In 1993 a very early version of Chart 'n' Art that disclosed a limited number of turtle-geometry-based Lisp expressions was tested on two subjects who were asked to create a few simple diagrams using the system. Subject A was a medical student with no programming experience, but significant familiarity with Macintosh and drawing programs. Subject B was a computer science post-doctoral fellow, also with Macintosh experience. Each subject was asked to follow a series of tasks involving first direct manipulation and then eventually Lisp programming. A talk-aloud protocol was used to elicit their thoughts during the session.

Subjects used the mouse to draw the silhouette of a tractor out of rectangles and ovals as shown in Figure 6-1. They then entered values into a spreadsheet which would be used to determine the positions of a series of small rectangles representing smoke coming out of the tractor exhaust. While performing these tasks, Lisp commands accumulated in a separate Transcript window (later called the Language window). Both subjects seemed to notice the appearance of the text in the window, but neither took particular interest in the meaning of the expressions. Subjects were then asked to use a programming language statement (`turn-south`) that had not been disclosed but that was similar to some of the expressions that appeared in the Transcript. The name of the command was given, but not its arguments, or the proper order for operations and operands. Nonetheless, Subject A was able to determine the appropriate syntax after inspecting the transcript. Subject B, the one with Lisp programming experience, surmised the expression without consulting the transcript.

Subjects were explicitly told about two more expressions, which again, because of the Chart 'n' Art's limited scope at the time, had not appeared in any disclosures. They were then asked to compose these expressions along with some that had been disclosed into a short program to map the values in the spreadsheet into rectangles positioned above the exhaust of the tractor. Both subjects made the important realization that they had already accomplished a similar shape placement task by direct manipulation when drawing the tractor itself. This time, both subjects studied the transcript window to deduce the shape creation command (`mouse-up 'STAMP`) and the command for specifying the location of the new objects (`go-absolute`). With mainly editing tips and minor syntactic help (such as when to "quote" symbols) from the experimenter, subjects were then able to combine these expressions with the ones that had been explicitly divulged to create the desired effect. The final task asked the subjects to find out about commands associated with two objects on the screen and write these down.

x	y
0	0
10	10
30	20
60	30

(a)



(b)

Figure 6-1. The target tractor drawing used in the feasibility study. The relative positions of the small rectangles coming out of the exhaust were determined by coordinates in a spreadsheet, as shown in (a).

Although subjects had initially been told how to inspect Lisp expressions related to drawing objects by double-clicking on the shapes, both had to be reminded of this technique.

The feasibility study provided encouraging results. Without explicit instruction, subjects noticed the one-to-one correspondence between direct manipulation commands and expressions appearing in the Transcript window and were able to make use of these disclosures to write Lisp expressions that actually solved drawing problems. On the other hand, the study also identified a number of critical interface and language design issues that would have to be resolved before users could learn from and employ disclosures effectively without assistance. For instance, could disclosures be designed so that users might attend to them more readily without prompting? Could the language be designed to be less ambiguous about when to quote symbols or perhaps avoid quoted symbols altogether? Could expressions related to the status of existing drawing objects be disclosed without explicit request (double-clicking) by users?

6.1.2 Preliminary Evaluation

A preliminary evaluation of a more complete implementation of the Lisp-based Chart 'n' Art was conducted in the summer of 1995. This study had two goals: (1) to gain feedback about the system design from users with varying programming background and (2) to help develop a methodology for formally measuring educational impact. Indeed, the study helped advance Chart 'n' Art's self-disclosure mechanisms by inspiring a complete rewrite of the software. Chapter 5 describes how the Chart 'n' Art interface evolved as a result of this evaluation and other feedback. The study's effect on the design of Chart 'n' Art's summative assessment is described in Section 6.2.

Five employees at Apple Computer, Inc. were recruited to participate in the study via company-wide email announcements. Four worked in software quality assurance (SQA) and the fifth was from Apple's research and development group. None of the subjects were regular programmers, but all had some programming experience with between 3 and 5 languages. Only the researcher had programmed in Lisp, the language being disclosed by Chart 'n' Art at the time.

Subjects first took a language pretest which asked them to describe the meaning of the following Chart 'n' Art Lisp expressions:

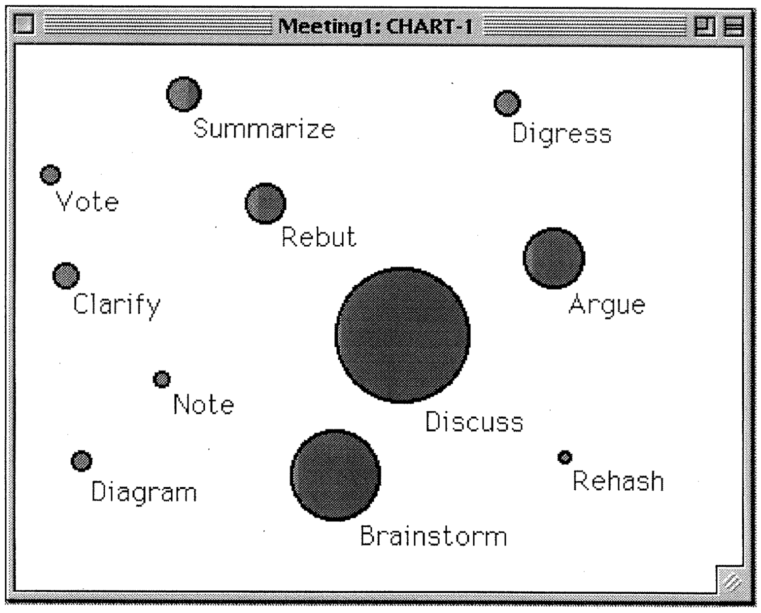
```
1.  (make-line :start-point '(20 10) :end-point '(30 35))
2.  (move Rectangle-3 '(20 40))
3.  (get-prop Rectangle-1 :size)
4.  (set-prop Oval-2 :fill-color *Green-Color*)
5.  (set-multiple (list Oval-2 Oval-3)
        :fill-color (list *Green-Color* *Red-Color*))
```

Answers were deliberately restricted to a single line of text below each question. Subjects were next acquainted with Chart 'n' Art through a brief tutorial that had them draw a rectangle, duplicate it, move both shapes, and select some cells in Chart 'n' Art's spreadsheet.

Subjects were next introduced to the bubble chart used by the CSCW researchers described Chapter 2. Recall that circles or "bubbles" represent activities in a meeting, the size of each circle proportional to the amount of time devoted to its corresponding activity. Subjects were then presented with the data in Figure 6-2a in a Chart 'n' Art spreadsheet window. They were asked to produce the corresponding activity chart using circles whose diameters (in pixels) were equal to the number of minutes spent in each activity. A sample solution is shown in Figure 6-2b, but note that the textual labels were not required. Throughout the charting task subjects were encouraged to talk aloud, try things out, and only ask questions of the experimenter if stuck. After 10-15 minutes, subjects were asked to read a one-page handout that described how to interpret the disclosures appearing in the Language window and how to reuse these disclosed expressions. Subjects were instructed to continue working on the charting task. After a total of 30 minutes, subjects were asked what they liked and disliked about Chart 'n' Art and then given a programming language posttest identical to the pretest. When filling out the posttest, subjects had access to their pretest answers.

Activity	Minutes
Argue	34
Brainstorm	51
Clarify	15
Diagram	12
Digress	15
Discuss	75
Note	10
Rebut	23
Rehash	8
Summarize	20
Vote	12

(a)



(b)

Figure 6-2. (a) Time spent in 11 different activities for a fictitious meeting that subjects were asked to represent with a bubble chart. (b) An example solution for the charting task. Note: subjects were only asked to draw the appropriately sized circles; no labeling was required, nor were the positions of the circles specified.

Of the five subjects, only one from SQA did not complete the task. All were able to use Chart 'n' Art's language to specify the exact size of one or more circles. However, each subject adopted a different strategy:

- One typed separate `make-oval` expressions with a list of two numbers for the size parameter, one for the height and one for the width, both equal to the number of minutes for each activity. Had she discovered the diameter parameter she would have needed to enter the number of minutes only once for each activity.
- The researcher, who had previously used Lisp, also did not see the diameter property and adopted a similar approach with the exception that within his make-oval expression he embedded calls to a spreadsheet function to automatically extract the number of minutes for each activity. This spreadsheet function had been disclosed earlier while he was browsing the spreadsheet.
- A third subject created a model oval, made 10 duplicates, and then used the `set-prop` command to adjust the diameter of each to correspond to the data.
- The fourth subject used a similar technique, except the duplicates were created one at a time.
- The remaining subject primarily used the mouse to draw each circle and adjust its size until the disclosure showed he had matched its size to the value from the minutes column of the spreadsheet. At the very end of the session, he began reusing the disclosed `resize` function and typing in values from the spreadsheet.

Four subjects answered the pretest and posttest short answer questions; the fifth (from SQA) ended the experiment before completing the posttest. Subjects' pretest answers were close to 100% correct with the exception of some confusion over question 5 regarding color mapping using the `set-multiple` command. Those who answered both pretest and posttest showed little change between tests. In fact, for all four respondents, answers to the posttest were almost word for word identical to the pretest. Only question 5 showed significant improvement, and this for only two subjects. One subject's answers to question 4 actually worsened.

All four subjects who answered the posttest questionnaire indicated that Chart n' Art disclosures helped them greatly to understand the system's programming language. Subjects did not agree on the value of the explanations for these disclosures. Three out of the four subjects felt the Language window would seldom get in the way of their work were they using Chart 'n' Art professionally. One of these commented that it might become a problem with screens smaller than 17 inches diagonal.

The variety of solutions to the charting task in this preliminary evaluation of Chart 'n' Art was in some ways an encouraging result. It demonstrated that the system could support a number of different approaches varying in their combination of direct manipulation and language-based interaction. Users did not see certain disclosures, for instance about the `diameter` property, but nonetheless learned enough from other disclosures to complete or partially complete the charting task. The strategies they adopted were not the most efficient; for example, no one used the infrequently-disclosed `set-multiple` command for operating on all circles at once. However, the few Lisp commands that subjects did learn about and use enabled them to draw more accurately, if not more rapidly, than with a traditional direct manipulation drawing tool. On the other hand, the variety of solutions made the evaluation of disclosure effects difficult, especially between subjects. Since each subject used a different sequence of mouse operations, they each were exposed to a different set of disclosures. It is not clear, then, whether the pattern of language use was the result of differing exposure to disclosures or individual differences in the ability to attend to and learn from disclosures.

Clearly, the language pretest and posttest failed to provide useful data on the effects of disclosure. The pretest proved to be too easy for all five subjects who, although they had never before seen Chart 'n' Art expressions, were able to successfully guess the semantics of almost every expression. Because of this "ceiling effect" it is no surprise that the posttests showed no significant improvement. The one consistent measure of disclosure effects would seem to be subjects' high opinion of the language learning opportunities provided by the Chart 'n' Art, indicated in the posttest questionnaire.

6.2 Goals for the Summative Assessment

The feasibility study and preliminary evaluation provided an invaluable formative assessment of Chart 'n' Art's self-disclosure mechanisms, leading to a number of non-trivial changes outlined in Chapter 5, including substituting SK8Script for Lisp as the system's end-user language. The preliminary evaluation in particular demonstrated that users could effectively employ disclosures to improve the accuracy of their drawings. However, due to both the design of the software being tested and the methodology used for this experiment, the study failed to directly address the question posed by the thesis of this report: Do well-designed self-disclosing systems have an educational impact on their users?

The summative assessment was designed to more carefully measure educational outcomes—specifically, in the case of Chart 'n' Art, to assess the nature of programming learning resulting from disclosures. In particular, the summative assessment sought to address three critical questions left unanswered by the preliminary evaluation:

1. ***The incidental learning question.*** What can users learn about an end-user programming language simply from exposure to disclosures, without actually employing them to execute expressions?

2. ***The composition question.*** How much can users learn about writing their own expressions and especially about composing nested or sequenced statements when exposed to short, isolated disclosures as typically found in Chart 'n' Art?

3. ***The programming experience question.*** What influence does programming background have on what can be learned from disclosures in Chart 'n' Art?

To examine the incidental learning question, the experiment included two groups exposed to disclosures: one that could use all of the Chart 'n' Art facilities for editing and reusing disclosed expressions, and that had full access to the SK8Script interpreter; and a second group that was limited to only seeing disclosures but not using them or any other SK8Script expressions. To address the composition question, the summative assessment exhibited two important differences from the preliminary evaluation. One, the protocol for the chart creation task encouraged and aided subjects in using programming as much as possible so that they might have a need for more complicated expressions. Two, a final task was added that required users to write a sequence of SK8Script statements, possibly involving nested expressions. Finally, to address the programming experience question, subjects' programming background was carefully classified prior to the laboratory studies and used in the analysis of results.

All three questions required testing instruments more sensitive to programming language gains than those used in the preliminary evaluation, prompting the design of a detailed multiple choice language test to be described in Section 6.2.2. The decision to use a multiple choice test led to a rather dramatic change in the experiment from a within-subjects design to a between-subjects design. This was done out of concern that a multiple choice pretest, in effect, could disclose elements of SK8Script to subjects, making it difficult to separate learning due to the pretest itself from learning due to Chart 'n' Art's disclosure mechanisms. Pretests are a standard component of vocabulary learning studies (c.f. Neuman & Koskinen, 1992) because correct answers and foils do little to teach subjects about other vocabulary words. However, tests involving syntactic and semantic questions such as the one described in Section 6.2.2 are difficult to construct without the answer's choices being inherently educational. A pretest made up of such answer choices could have a significant impact on an identical or even similar posttest, as long as subjects could remember inferences made in the pretest. For the summative assessment, then, a control group which did not see disclosures was used to establish a baseline test score rather than giving the same subjects both pretest and posttest.

6.2.1 *Distinguishing Training and Performance Tasks*

Another major change from the preliminary evaluation was in the design of the experimental tasks. Rather than having a single task such as the activity diagram both provide subjects with programming language learning opportunities and at the same time measure subjects' programming language acquisition, the summative assessment made a clear distinction between training tasks and performance tasks. The former were designed to expose subjects to a consistent set of disclosures that would be useful later in the experiment. The latter were employed to assess what language skills had been learned in training and to measure the degree to which disclosures facilitated the writing of programming statements.

6.2.2 *More Sensitive Testing Instruments*

The last important change from the preliminary evaluation was in testing instruments. An additional three-part short-answer question was devised to assess more general SK8Script programming knowledge. Furthermore, a multiple choice programming language test covering vocabulary, syntax and semantics was constructed to replace the simple short-answer questions in the preliminary evaluation that subjects found too easy. An example of one of the multiple-choice questions appears in Figure 6-3.

Almost every question in the multiple-choice test was designed to meet the following criteria:

1. The answer could be inferred after observing one or more relevant disclosures.

2. The relevant disclosures would likely be shown during the training task.

3. The answer was not obvious to subjects without training.

4. The question itself tested important knowledge, not language trivia.

5. The answer choices minimized clues that might help answer other questions.

25. Which expression will **draw a yellow rectangle with a blue border**?

☐ a. `make Rectangle {fillColor: Yellow, frameColor: Blue}`

☐ b. `make Rectangle whose fillColor = Yellow and whose frameColor = Blue`

☐ c. `make Rectangle with fillColor Yellow with frameColor Blue`

☐ d. `set frameColor Blue of set fillColor Yellow of make Rectangle`

Figure 6-3. A syntax question from the multiple choice programming language test.

To help satisfy criteria 1 and 2, pilot tests were used to identify the set of disclosures subjects were likely to encounter during the training tasks. Pilot testing by subjects who saw no disclosures also helped ensure answers were difficult to guess without training. To address criterion 4, the programming walkthrough of the performance task along with less formal techniques was used to identify critical pieces of programming knowledge. Questions that asked subjects to discriminate between vocabulary with only slight spelling differences or syntax with only minor order differences were avoided.

The fifth criterion concerning minimizing interactions between questions was more difficult to satisfy. For instance, consider the programming language facts that one can infer by simply reviewing the answers a-d in Figure 6-3, regardless of one's knowledge of the correct answer (c):

1. There is a token called `Rectangle` which presumably refers to a rectangle shape.

2. There are tokens called `Yellow` and `Blue` which presumably refers to colors.

3. The language uses a fairly English-like syntax with perhaps a few symbols such as '`=`', '`{`', and '`:`'.

4. The token 'make' is likely the first word in a valid shape creation command.

Although these kinds of clues were difficult to eliminate, careful ordering of test questions helped reduce the influence one question had on another within the same test. For instance, the syntax question 25 in Figure 6-3 was placed after vocabulary questions about `Rectangle` and `Yellow`. To make this ordering effective, subjects were, of course, prohibited from reviewing previously answered questions in the multiple-choice test.

6.3 Method

6.3.1 Subjects

Subjects were recruited via fliers posted throughout the University of Colorado campus. Candidates were interviewed over the phone to ensure they had used a Macintosh drawing program in the last four years and had not used the SK8Script programming language. The phone interview also asked subjects to rate their level of expertise in eight Macintosh drawing programs and 23 programming languages. Pilot tests helped make these two lists as complete as possible; however, subjects could also rate their level of expertise in drawing programs and programming languages missing from either list. All ratings used the following four standard response categories: "very experienced," "somewhat experienced," "not very experienced," and "not at all experienced." The complete set of interview questions are reproduced in Appendix B.

Because of concern that programming experience might strongly influence educational impact, subjects were classified under three experience groups: no familiar programming languages (E0), familiarity with one or two languages (E1-2), and familiarity with three or more languages (E3+). Subjects were considered to be familiar with a language if they indicated that they were "very experienced," "somewhat experienced," or "not very experienced" for any language mentioned in the phone interview. Only answers of "not at all experienced" were considered an indication of lack of familiarity. The use of the known-languages statistic as a measure of programming background was suggested by studies which indicated that number of languages was the best predictor of programming ability among a variety of variables, including years of experience (Sheppard, Curtis, Milliman & Love, 1979).

As subjects qualified for the experiment, they were scheduled and randomly assigned to one of three conditions —no training, passive training, or active training—balanced across programming experience. Randomization was accomplished through a combination of hat-drawing and computerized random number generation. The phone interviews and condition assignments were conducted independently by assistants, so that the experimenter would be blind to subjects' recorded programming experience throughout the laboratory study.

Of the 45 people who were scheduled, 33 actually participated in the study. Each was paid $45. After omitting 3 subjects because of experimenter error and problems with the materials, 30 were finally used in the data analysis: 19 undergraduates, 10 graduate students, and one person who received her undergraduate degree the semester before the study. Due to scheduling difficulties, only 7 subjects from the E3+ category were actually tested, while 11 and 12 subjects participated in E0 and E1-2, respectively. Table 6-1a illustrates the distribution of programming experience across conditions. For data analysis, subjects were reclassified using two experience categories instead of three so as to increase the numbers of participants per group and thereby improve the likelihood of statistically significant results. The low experienced group, E0-1, represented subjects who were familiar with at most one language, while the high experience group, E2+, represented subjects familiar with two or more languages. Table 6-1b shows the distribution of subjects across conditions using these two programming experience categories.

6.3.2 Materials

To assess baseline programming and drawing program experience, each subject was initially interviewed by phone using the form described in Section 6.3.1 (Appendix A). In the laboratory, subjects began filling out a demographic questionnaire (Appendix B) which consisted of multiple choice questions seeking additional background information including attitudes towards programming. Subjects were then asked to perform a series of training tasks, (to be described in more detail in the Section 6.3.3.) Subjects were guided through these tasks by verbal instructions from the experimenter so that participants could keep their eyes on the screen and attend to disclosures.

Table 6-1. (a) The distribution of programming experience over experimental conditions where E0 is the least experienced group and E3+ the most experienced. These categories were used when assigning subjects to conditions. (b) The distribution of programming experienced using only two experience categories. These categories were used for data analysis.

	Condition			
Experience Category	No Training	Passive Training	Active Training	Total
E0	3	4	4	11
E1-2	4	4	4	12
E3+	3	2	2	7
Total	10	10	10	30

(a)

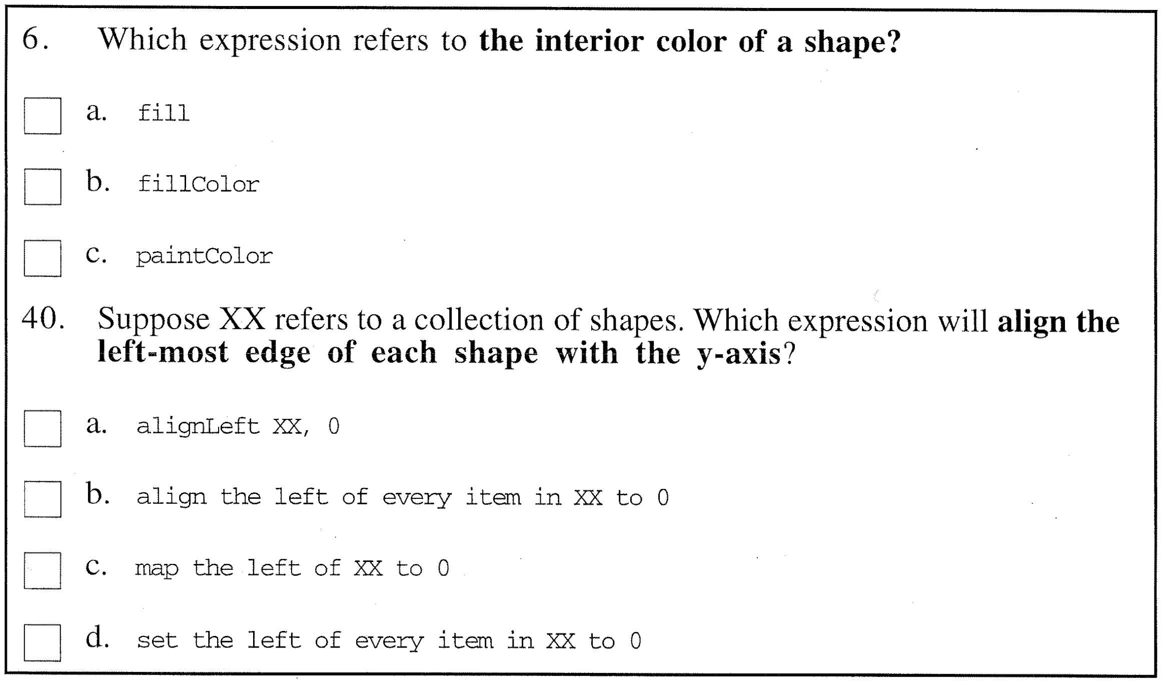
	Condition			
Experience Category	No Training	Passive Training	Active Training	Total
E0-1 (low)	5	5	7	17
E2+ (high)	5	5	3	13
Total	10	10	10	30

(b)

A short-answer and multiple-choice programming language test were used to assess programming language understanding. The short-answer questions were administered on paper (see Appendix F), while the multiple choice test required that subjects fill out an on-screen computer form (reproduced in Appendix G). Identical tests were used for all three conditions. The short-answer test asked subjects to describe the operations and operands in Chart 'n' Art's language and the syntax for combining these into sentences. In the instructions for the short-answer test, definitions for "operation" and "operand" were provided which drew on analogs with English grammar. Typical questions in the multiple-choice test consisted of a sentence in English followed by three or four SK8Script answer choices. For the purpose of analysis, questions were divided into vocabulary, syntax, and semantic questions. Examples of vocabulary and semantic questions appear in Figure 6-4 and a syntax question is shown in Figure 6-3.

To test language production abilities, subjects were also given three performance tasks (described in detail in Section 6.3.3) which involved programming in Chart 'n' Art. For each task, subjects were shown printouts of the target chart shown in Figure 6-5b and a variation on this same chart that used two different colored bars. For the first two tasks, subjects could also consult the 64-page section from the SK8 manual which described the SK8Script language, although subjects were told only to use this as a "last resort."

A custom version of Chart 'n' Art was built for the experiment which provided all of the standard disclosure mechanisms but disabled certain features to ensure the presentation of disclosures would be consistent within individual sessions and across subjects. Windows were locked into an arrangement similar to that in Figure 4-1, with the Language window occupying approximately a third of the screen space. These windows could not be hidden, closed, or moved, nor could users adjust the magnification of the drawing window. Other features were added to Chart 'n' Art to simplify the experiment. For example, a special Data menu was provided to select from the data sets used in the training and performance tasks. Data appeared in a small text window in the upper right corner of the screen with each value separated by a comma. Subjects could copy a single value or a string of values from this data window directly



6. Which expression refers to **the interior color of a shape?**

☐ a. `fill`

☐ b. `fillColor`

☐ c. `paintColor`

40. Suppose XX refers to a collection of shapes. Which expression will **align the left-most edge of each shape with the y-axis**?

☐ a. `alignLeft XX, 0`

☐ b. `align the left of every item in XX to 0`

☐ c. `map the left of XX to 0`

☐ d. `set the left of every item in XX to 0`

Figure 6-4. Two more questions from the multiple-choice test. Number six was considered a vocabulary question, number forty a semantic question.

into an expression being typed in the Language window using the standard Mac OS Edit menu items. Additionally, a logging feature was added to Chart 'n' Art which automatically created detailed records of disclosures and user actions to support later analysis.

Chart 'n' Art was run on a Apple Macintosh Quadra 840AV with 40 MB of RAM. Screen capture hardware was used in combination with a VHS-VCR to record all on-screen user interaction. This video recording arrangement limited the screen resolution to 640x425 pixels. Although a 14-inch diagonal monitor would have been the best match for this resolution, only a 20-inch multiscan monitor was available for the experiment. To mitigate the "chunkiness" of the resulting images and text, the screen was set back an additional 10 cm or so from the keyboard.

6.3.3 Procedure

The study involved three conditions: no training, passive training, and active training. Each condition included three phases: (a) a training phase, (b) a memory assessment phase involving a written and multiple choice test, and (c) a performance phase involving the creation of a target chart using SK8Script as much as possible. The sequence of instructions, tasks, and tests for each phase of the experiment are summarized in Table 6-2. Subjects were tested individually over a single session lasting approximately two to three hours: one hour for the training phase, ten minutes for the memory assessment phase, a ten minute break, and the remaining time for the performance phase.

Training Phase. The training phase consisted of nine events. The first two events oriented subjects to Chart 'n' Art's direct manipulation interface—its windows, palettes, and menus—and provided basic instructions and how to complete the training exercises. Subjects then began the first of the five training tasks in Table 6-3. This first task, T1, involved creating a combination column and line chart and served to

Table 6-2. The sequence of instruction, task, and test events for each condition in the formal experiment. From top to bottom, the dotted lines demark the training phase, memory assessment phase, and performance phase, respectively.

Event	Type	No Training	Passive Training	Active Training
1	Instr.	intro. to d.m. interface	intro. to dm. interface	intro. to d.m. interface
2	Instr.	n/a	told to ignore disclosures	told to watch disclosures
3	Task	T1	T1	T1
4	Instr.	n/a	intro. to disclosure mechanisms	intro. to disclosure mechanisms
5	Instr.	n/a	n/a	how to edit & evaluate expressions
6-9	Task	T2-T5	T2-T5	T2-T5
10	Test	short answer test	short answer test	short answer test
11	Test	multiple-choice test	multiple-choice test	multiple-choice test
12	Instr.	intro. to disclosure mechanisms	intro. to disclosure mechanisms	intro. to disclosure mechanisms
13	Instr.	how to edit & evaluate expressions	how to edit & evaluate expressions	how to edit & evaluate expressions
14-16	Task	P1-P3	P1-P3	P1-P3

incrementally introduce subjects to many of Chart 'n' Art's direct manipulation tools and menus. The fourth and fifth events in the training phase provided subjects with additional instructions.

The last four events consisted of the training tasks T2 through T5 where T2 through T4 were more simple column chart tasks designed to introduce other language features such as for duplicating shapes multiple times. The final training task, T5, was to produce a thirty-item column chart—an attempt to motivate the subjects who had access to SK8Script to try the language if they had not done so already.[12] For the subjects who were privy to programming language disclosures, the entire sequence of training tasks was designed to uncover the fundamental programming constructs users would need for the performance tasks. These fundamental constructs were identified through a programming walkthrough (Bell et al., 1994) of two possible solutions to the chart creation problems posed by the performance tasks.

Training Phase Treatments. The training phases of the three conditions differed in certain essential ways. First, Chart 'n' Art was set up to behave differently for each group. For the no training condition, mouse disclosures and selection labeling were disabled and the Language window was made invisible; thus subjects in this condition saw no disclosures whatsoever. Subjects in the passive training condition could see disclosures appearing in the Language window and other places on the screen, but

[12] Pilot studies indicated that thirty items involved repetition beyond the threshold of patience for many subjects, who would often wonder out loud if there was an easier way to do the task than by dragging out each shape.

Table 6-3. A brief description of the five main training tasks. Note that subjects were given more detailed instructions in the actual experiment as described in Appendix C.

Task	Description
T1	Create a combination column and line chart with three data points each (provided by the experimenter). Adjust the dimensions of the columns slightly. Add more columns to the chart by duplicating existing ones. Add a background.
T2	Create a column chart representing the values 65, 80, 112, 85, and 92. Start with a single 10-unit wide column and duplicate it to create the rest of the columns.
T3	Adjust the heights of the columns to correspond to the values 49, 119, 78, 108, and 38. Change the width of each column from 10 to 20.
T4	Adjust the heights of the columns to correspond to the values 88, 131, 29, 52, and 85. Change the width of each column from 20 to 15.
T5	Create a column chart representing the 30 values 85, 112, 130, 98, 93, 135, 69, 83, 123,107, 36, 82, 83, 101, 130, 110, 60, 56, 92, 55, 67, 99, 45, 46, 41, 69, 126, 102, 133, and 108. Start with a single 10-unit wide column and duplicate it to create the rest of the columns.

could not edit or execute any of the disclosed expressions. Finally, subjects in the active training condition could not only see all disclosures, but reuse them at will through the Language window.

The second difference in the training phase was that the experimenter gave different instructions to each group. Only the passive and active training conditions received the instructions in events 2 and 4, and only the active training condition received the instructions in event 5. In event 2, the experimenter initially identified the Language window by reading the following description:

> *The Language window is where Chart 'n' Art presents information to you about its programming language, SK8Script. This programming language can among other thing help automate the creation of our chart and place chart elements more accurately.*

Subjects in the passive training condition were then told, "For now, don't worry about programming language feedback." In contrast, subjects in the active training condition were told, "Pay attention to the programming language information which shows up here and other places on the screen. You'll be using it soon." In event 4, both the passive and active conditions were walked through a brief series of exercises (see Appendix D) introducing each of Chart 'n' Art's disclosure mechanisms. These exercises were not intended to teach programming expressions to subjects, but rather to help them interpret the connection between their activity and the disclosures appearing on the screen. Finally in event 5, subjects in the active condition were shown how to reuse and edit disclosures and execute expressions in the Language window. Only these subjects were then encouraged at the onset of the remaining four training tasks to "Try and use the language as much as possible to make things exact and to help automate repetitive tasks." Subjects in the active training group were guided when necessary by a set of predetermined hints, detailed in Appendix E. The hints also helped minimize frustrating breakdowns observed in pilot studies of the summative assessment.

Memory Assessment Phase. The memory assessment phase consisted of two events: event 10, the short-answer test to evaluate subjects' ability to recall the operations and operands used in Chart 'n' Art version of SK8Script; and event 11, the multiple-choice test to evaluate subjects' ability to recognized valid SK8Script expressions. Unlike the previous phase, all three groups were given identical instructions and materials. They could not use Chart 'n' Art during these tests.

Performance Phase. The performance phase consisted of five events. In the first two events, 12 and 13, the experimenter stepped all subjects through the same disclosure exercises given to the active and passive training groups in event 4 and the active group in event 5. For subjects in the passive and active training conditions, these exercises were therefore at least partial review from the training phase. The idea was to ensure that all subjects were, at this point, familiar with interpreting, editing, and reusing disclosures.

The remaining events in the performance phase consisted of the performance tasks P1 through P3 (listed in Table 6-4) which involved creating a floating bar chart similar in spirit to the temperature chart in Chapter 2. Recall that a floating bar chart consists of two data series, one corresponding to the bottoms of a vertical bar, and the other to the tops. For the summative assessment, the first data series consisted of the elevations of 30 thirty cities in a fictitious country and the second series consisted of the heights of each of these building (measured from the base of each building to its top). The actual data used in the study is shown in Figure 6-5a with the target chart illustrated in Figure 6-5b. Performance task P1 asked subjects to generate the floating bar chart and the next task, P2, was to highlight the 16 buildings that were thirty units tall or higher by coloring them differently from the rest of buildings. Although not advertised to subjects, performance tasks P1 and P2 were allotted a maximum of 45 minutes and 10 minutes respectively.

Table 6-4. A brief description of the three main performance tasks. Note that subjects were given more detailed instructions in the actual experiment.

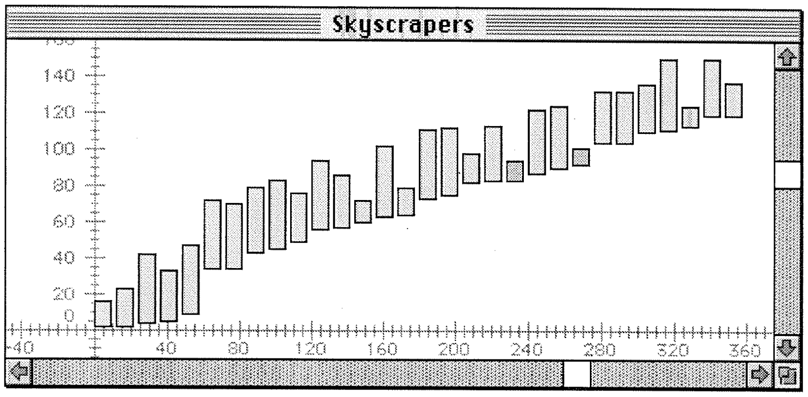
Task	Minutes Allotted	Description
P1	45	Create a floating bar chart to depicting the 30 buildings in the "skyscraper" data. (See Figure 6-5.) Start with a single 10-unit wide column and duplicate it to create the rest of the columns.
P2	10	Highlight all of the buildings at least 30 units tall by coloring their columns red.
P3	10	Type a program from scratch that would reproduce the target skyscraper chart, including the proper highlighting of tall buildings. Assume that you are starting with a blank drawing window.

In an effort to avoid the variety of solutions demonstrated in the preliminary evaluation and simplify between-subject comparisons, users' explorations throughout performance tasks P1 and P2 were guided when necessary by the same hints (see Appendix E) which had been available to the active training condition during tasks T1 through T5. It is also important to note that for these two tasks, all subjects used the full-featured Chart 'n' Art setup provided only to those in the active training condition during the training phase. In other words, even subjects in the no training and passive training conditions saw disclosures appearing in the Language window and in other places on the screen. All subjects were also encouraged to edit and reuse disclosures.

```
30 city elevations
  2,  2,  4,  5,  9,
 34, 34, 43, 45, 49,
 56, 57, 60, 63, 64,
 73, 75, 82, 83, 83,
 87, 90, 92,104,104,
110,111,113,119,119

size of the tallest
building in each city
15, 22, 39, 29, 39,
39, 37, 37, 39, 28,
39, 30, 13, 40, 16,
39, 38, 17, 31, 12,
36, 35, 10, 29, 29,
27, 40, 12, 32, 19
```

(a)



(b)

Figure 6-5. (a) The data used for the column charts in the performance tasks. (b) The target chart for the performance tasks.

For the final performance task, P3, all history items in the Language window were erased and all other windows in Chart 'n' Art were disabled. Subjects were instructed to simply type into the Language window their best guess of the program which would recreate the target chart for the previous task, P2, assuming a blank drawing window. Subjects were told that the essential aspects for the program were generating the shapes, and positioning, sizing, and coloring them. Details about the exact placement or sizing of elements were to be fleshed out only if time allowed. Subjects were encouraged to make use of the full ten minutes allotted for the task.

6.3.4 Coding and Reliability

Subjects' responses to the short answer questions were coded in terms of (a) the degree they demonstrated an understanding of the SK8Script programming language, and (b) the degree they demonstrated an understanding of the questions themselves, i.e. about the concepts of operations and operands. Understanding was scored using the following six-point scale: (1) no understanding, (2) a little understanding, (3) some understanding, (4) considerable understanding, (5) a lot of understanding, and (6) complete understanding. Scores for the short-answer questions were coded independently by two judges with SK8Script programming experience. Both judges were blind to the training conditions and one was not even aware of the nature of the conditions used in the experiment. Language understanding scores between judges had a correlation coefficient of .67. For problem understanding scores, the correlation coefficient was .44. Subjects' final programs from performance task P3 were coded by the same judges using the same scale to measure understanding of the SK8Script programming language. Again, the judges were blind to training conditions. For the program solutions they evaluated, the correlation coefficient between judges was .79

6.4 Results

This section summarizes results from the summative assessment through a number of different quantitative and qualitative perspectives. The intent is to enable a discussion of both scientific issues—focused on statistically significance learning outcomes resulting from disclosures—as well as "engineering" issues (Landauer, in press)—concerned with the costs and benefits of self-disclosure mechanisms. Let us start by examining the mean scores for the three tests which revealed significant differences between conditions: the short-answer test, multiple-choice test, and final program. Table 6-5 lists the means grouped by condition and programming experience and Table 6-6 shows the overall statistically significant results. More detailed tables of results appear later in this chapter and in Appendix H.

Table 6-5. Mean scores for the short-answer test, multiple-choice test, and final program grouped by condition and programming experience. The short-answer test scores are on a subjective six-point scale (described in 6.4.1), while the multiple-choice test scores are percent correct.

	No Training	Passive Training	Active Training	Overall
Short Answer	1.45	2.00	3.20	2.22
Low Exper.	1.50	1.40	3.36	2.24
High Exper.	1.40	2.60	2.83	2.19
Multiple Choice	34%	41%	53%	43%
Low Exper.	40%	35%	55%	44%
High Exper.	28%	47%	49%	40%
Final Program	3.00	2.40	3.45	2.95
Low Exper.	2.70	2.00	3.36	2.77
High Exper.	3.30	2.80	3.67	3.19

Table 6-6. Summary of statistically significant pair-wise differences between conditions generated using Scheffé post-hoc tests of ANOVAs of short-answer, multiple-choice, and final program scores. Each significant difference is described by its mean, the 95% confidence level, and the p-value. Low experience subjects are those familiar with at most one programming language (E0-1). High experience subjects are those familiar with at least two languages (E2+).

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Short Answer	1.20	0.94	.0106	1.75	0.94	.0003			
Low Exper.	1.96	0.96	.0003	1.86	0.96	.0005			
High Exper.									
Multiple Choice				19%	16%	.0150			
Low Exper.	20%	15%	.0104	15%	15%	.0489			
High Exper.									
Final Program	1.05	1.02	.0432						
Low Exper.	1.36	1.14	.0198						
High Exper.									

The table shows the mean, 95% confidence intervals, and p-values for the pair-wise differences between short-answer, multiple-choice, and final program scores in each condition. Performance measures for tasks P2 and P3 are omitted from the table, since, as discussed in Section 6.4.3, the data were largely inconclusive. The table also breaks these differences down by subjects' programming background using the experience classification developed in Section 6.3.1. Empty cells in the table represent differences that were not statistically significant. Note that there were statistically significant differences in pair-wise tests between the active and passive training conditions and the active and no training conditions, but not between the passive and no training conditions. Also note the lack of significant differences among subjects with high programming experience, as well as the lack of differences in the final program scores between the active and no training groups.

6.4.1 Short-Answer Scores

Table 6-7 lists the mean understanding scores for the short-answer test, grouped by condition. The average subject in the no training group demonstrated somewhere between "no" and "a little" understanding of SK8Script and its operations and operands, the average passive training subject showed "a little" understanding, and the average active training subject exhibited between "some" and "considerable" understanding. However, the only pair-wise comparisons between conditions that were significant in an analysis of variance (ANOVA) ($F_{2,27} = 10.907$) were between the active training and no training conditions ($p < .0003$) and between the active and passive training conditions ($p < .0106$).

Table 6-7. Degree of programming language understanding demonstrated in the short answer test. The understanding score is the mean understanding per condition as rated by judges on a scale from 1 to 6. For each condition the table lists mean score, the 95% confidence level, standard error, and standard deviation.

	No Training	Passive Training	Active Training	Overall
Understanding Score	1.45	2.00	3.20	2.22
Conf. Level	0.36	0.61	0.79	
Std. Err.	0.16	0.27	0.35	
Std. Dev.	0.50	0.85	1.11	
Low Exper. Mean	1.50	1.40	3.36	2.24
Conf. Level	0.76	0.68	0.58	
Std. Err.	0.27	0.25	0.24	
Std. Dev.	0.61	0.55	0.63	
High Exper. Mean	1.40	2.60	2.83	2.19
Conf. Level	0.52	0.81	5.02	
Std. Err.	0.19	0.29	1.17	
Std. Dev.	0.42	0.65	2.02	

Because of the high variability in the active training condition, a t-test[13] was used to reassess the significance of the 0.55 point difference between the scores in the passive training and no training conditions. (Detailed results from this and other t-tests appears in Appendix H.) Nevertheless, no significant difference was found ($p < .0943$). Additional analysis was conducted to determine "odds-ratio" (Landauer, in press) for this difference. The results indicate that the odds are better than 9:1 (using the t-test) that the mean difference between the passive and no training scores is between 0.01 and 1.09.

In an effort to further clarify the weak differences observed in the short answer scores between the no training and passive groups, their answers were examined for signature phenomena. Were there consistent errors made by one group and not the other? A

[13] A t-test excludes influence from the variability in the groups left out of the pair-wise comparison.

review of answers revealed no characteristic differences. Although many in the no training group erroneously described Chart n Art's language as sequences of mouse-actions, so did four in the passive training group. Additionally, three in the no training group described a plausible text-based language not unlike SK8Script.Answers by some subjects indicated that they did not fully understand the short-answer questions, possibly because of confusion over the mathematical terms "operation" and "operand." For instance, one subject's description of SK8Script operations was simply "gives you exact location of points." Another subject seemed to confuse direct manipulation actions with textual programming language commands, claiming that an example of a SK8Script command was "click on rectangle and drag it up with mouse." To check that such subjects did not distort the short answer results, the data was reanalyzed after removing the scores from the five subjects (distributed approximately evenly across conditions) who judges rated as having little or no understanding of the questions. The results were very similar to the original analysis with no training, passive training, and active training having mean scores of 1.44, 2.13, and 3.28, respectively. There was no change in significant differences between groups.

Figure 6-6 illustrates the differences among understanding scores as grouped by programming background. As shown in Table 6-6, the pair-wise significant differences in an ANOVA ($F_{2,14} = 20.765$) were (a) between low experience subjects in the active condition (A_L) and low experience subjects in the passive condition (P_L) ($p < .0003$), and (b) between A_L and low experience subjects in the no training condition (N_L) ($p < .0005$). For subjects with high programming experience, an ANOVA between the no training group (N_H), the passive training group (P_H), and the active training group (A_H) showed no significant differences ($F_{2,10} = 2.462$, $p < .1351$). However, among the high-experience subjects, a t-test between the no training and passive training group (leaving out the highly-variable active condition scores) was statistically significant ($p < .0085$).

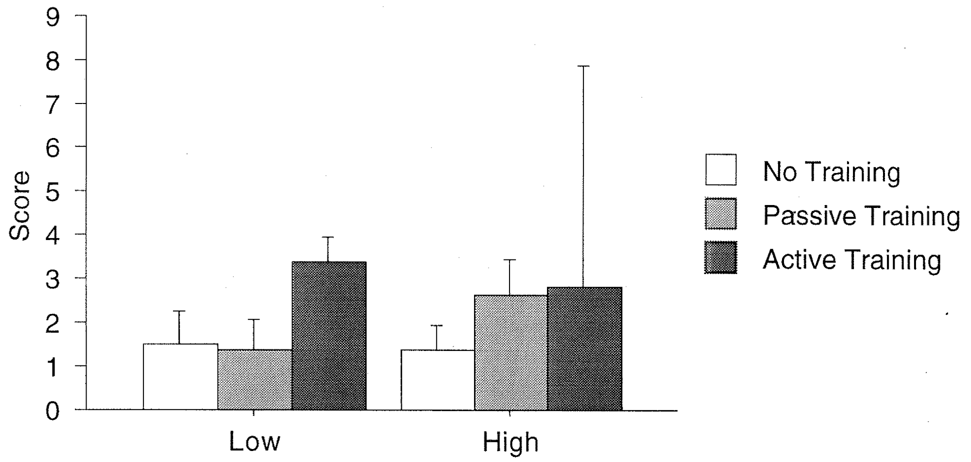


Figure 6-6. Understanding scores for the short answer test grouped by condition and programming experience. The error bars indicate the 95% confidence levels.

Figure 6-6 suggests a potential interaction effect between condition and programming experience. It appears that scores for the passive training subjects increased from low experience to high, while scores for the active training subjects decreased. An ANOVA of interaction between condition and experience ($F_{2,27} = 2.932$, $p < .0726$) showed no statistically significant effects, but its low p-value warranted closer inspection of the data. The combined scores from P_L and A_H were compared with the combined scores from P_H and A_L in a t-test which did show a significant difference ($p < .0299$). Of course, one must be cautious not to overvalue this statistic, considering the particular comparison was suggested only *after* a review of the data.

6.4.2 Multiple Choice Scores

The first two questions on the multiple choice test asked subjects to subjectively rate their programming language learning experience using Chart 'n' Art. Table 6-8 and Table 6-9 show the distribution of responses for these two questions. The first question asked how helpful subjects found Chart 'n' Art's programming language instruction and the second asked subjects to rate how much programming they learned. Note that even those who received no language instruction tended to rate their learning experience positively.

Table 6-8. Responses to the subjective question: "How helpful did you find Chart 'n' Art's programming language instruction?"

	No Training	Passive Training	Active Training	Total
not at all helpful (1)	1	0	0	1
not very helpful (2)	2	2	1	5
somewhat helpful (3)	4	7	4	15
very helpful (4)	3	1	5	9
Mean	2.90	2.90	3.40	3.07

Table 6-9. Responses to the subjective question: "How much do you think you learned from Chart 'n' Art's programming language instruction?"

	No Training	Passive Training	Active Training	Total
not at all (1)	1	0	0	1
not very much (2)	3	3	0	6
some (3)	5	7	6	18
a lot (4)	1	0	4	5
Mean	2.60	2.70	3.40	2.90

The remaining 39 multiple choice questions measured subjects' understanding of SK8Script vocabulary, syntax, semantics or some combination of these topics. A total of 19 questions addressed vocabulary, 19 syntax (after dropping one because of an error), and 5 semantics. Vocabulary questions provided three answer choices, while all other questions offered four possibilities. Table 6-10 lists the overall scores for the multiple-choice test, grouped by condition. Figure 6-7 illustrates the mean percent correct for each question topic for each condition. Finally, Table 6-11 summarizes the statistically significant differences between conditions. Detailed results from the multiple-choice test appear in Appendix H.

Table 6-10. Overall percent correct of multiple choice programming questions for each condition, along with the adjusted scores after removing the answers to 11 questions that may have been influenced by the introduction to Chart 'n' Art's disclosure mechanisms. For each mean the table lists the 95% confidence, standard error, and standard deviation.

	Raw			*Adjusted*		
	No Training	Passive Training	Active Training	No Training	Passive Training	Active Training
Overall Mean	34%	41%	53%	30%	33%	44%
Conf. Level	10%	10%	10%	10%	7%	11%
Std. Err.	0.043	0.043	0.045	0.044	0.033	0.048
Std. Dev.	0.136	0.135	0.143	0.140	0.103	0.151
Low Exper. Mean	40%	35%	55%	36%	28%	45%
Conf. Level	12%	8%	10%	13%	5%	11%
Std. Err.	0.044	0.029	0.042	0.048	0.019	0.045
Std. Dev.	0.098	0.066	0.110	0.106	0.043	0.119
High Exper. Mean	28%	47%	49%	24%	37%	41%
Conf. Level	19%	21%	57%	19%	16%	60%
Std. Err.	0.069	0.074	0.133	0.069	0.058	0.140
Std. Dev.	0.155	0.166	0.230	0.154	0.129	0.242

Notice in Table 6-6 that the only significant difference in overall multiple-choice scores was the 19% gap between the active and no training groups ($F_{2,27} = 4.996$, $p < .0150$). When scores were broken down by topic as shown in Figure 6-7, it appeared that syntax questions had a large influence on the difference. Indeed, a Scheffé post-hoc test of the pair-wise differences between conditions for each question topic (Table 6-11) indicated a significant difference between active and no training for syntax questions.

Table 6-11. Summary of statistically significant differences across the three main multiple-choice question topics. The pair-wise differences were generated using Scheffé post-hoc tests of ANOVAs for each topic. Each significant difference is described by its mean, the 95% confidence level, and the p-value.

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Vocabulary									
Syntax				30%	22%	.0053			
Semantics							0	•	•

Again, because of the high variability in the active training condition, a t-test was used to reassess the significance of the 16% difference between the syntax scores in the passive training and no training conditions. No statistically significant difference was found ($p < .0758$). However, from an odds-ratio perspective, the results suggest that the chance is better than 9:1 (using the t-test) that the mean difference between the passive and no training scores is between 1% and 30%. A t-test was also used to compute an odds-ratio for the relatively large (but not significant) difference between overall scores of the active and passive groups. In this case the results indicate that the odds are better than 9:1 that the mean difference between these two groups is between 2% and 23%.

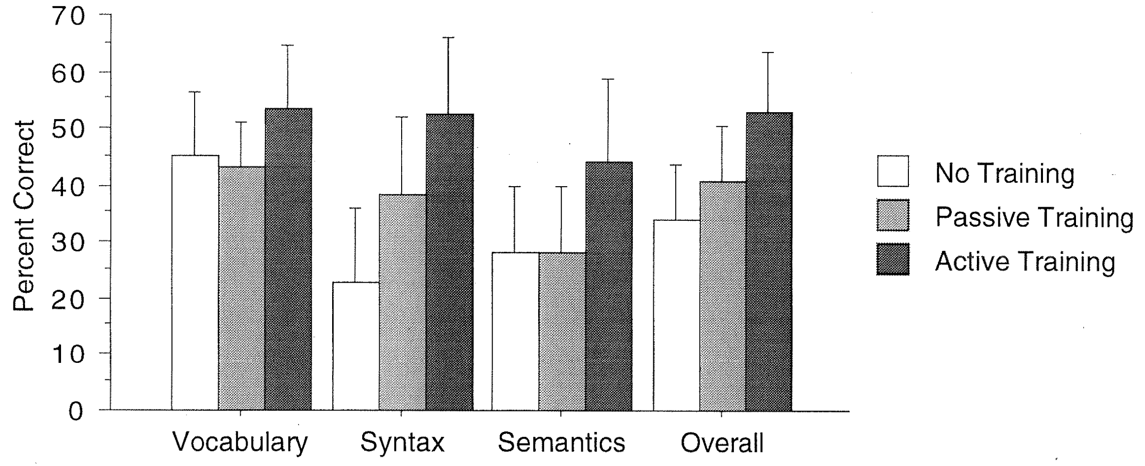


Figure 6-7. Percent correct of multiple choice programming questions on the topics of vocabulary, syntax, and semantics, and overall percent correct, broken down by condition. The error bars indicate the 95% confidence levels.

Figure 6-7 suggests the possibility of an interaction effect between condition and question topic, since it appears that passive training may have only helped subjects perform better on syntax questions when compared to no training subjects. However, an ANOVA of interaction between condition and question topic ($F_{4,25} = 1.261$, $p < .2923$) indicated that the effects were insignificant.

A concern with the experimental method was that subjects in the passive condition might have *actively* learned some SK8Script during the brief introduction to Chart 'n' Art's disclosure mechanisms (event 4) which occurred approximately 30 minutes prior to the multiple choice test. Additional analysis was conducted to determine what, if any, influence these exercises might have had on the results in Figure 6-7. A review of the disclosure exercises indicated that the answers to as many as 11 questions in the multiple-choice test might have been divulged by disclosures during the exercises. It should be noted that each suspect disclosure appeared only once during this time. Removing these questions from the data reduced the percent correct for all groups as shown in the adjusted scores in Table 6-10. However, none of the significant differences between groups for either the overall scores or those broken down by question topic was affected.

Figure 6-8 illustrates the differences seen when the multiple-choice scores are grouped by programming background. The only pair-wise comparisons between conditions that were significant in an ANOVA ($F_{2,14} = 7.394$) were (a) the 20% difference between A_L and P_L and (b) the 15% difference between A_L and N_L. Like the corresponding figure for understanding scores, Figure 6-8 suggests a possible interaction effect between condition and programming experience where the scores appear to increase from P_L to P_H, but decrease from A_L to A_H. Again, an ANOVA of interaction between condition and experience ($F_{2,27} = 2.932$, $p < .1592$) showed no statistically significant effects. However, since the ANOVA indicated that the odds were better than 5:1 that there was an interaction, further analysis was conducted. The combined scores from P_L and A_H were compared with the combined scores from P_H and A_L in a t-test which did not show a significant difference ($p < .0956$).

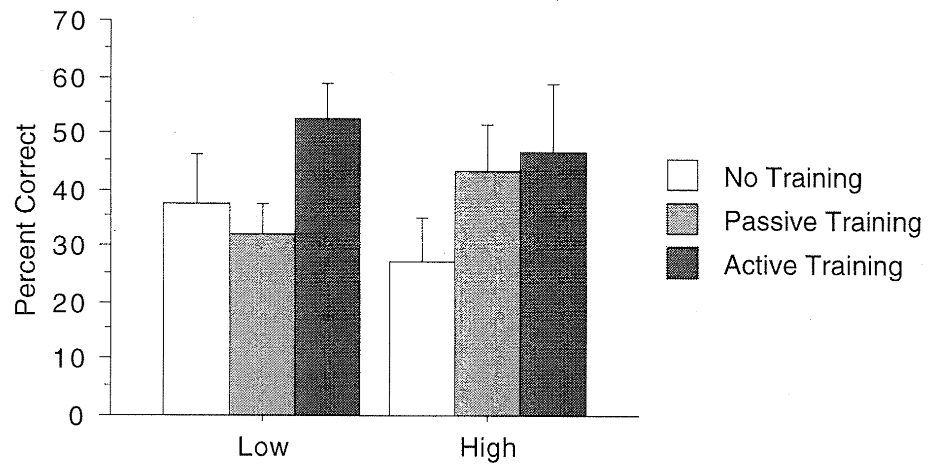


Figure 6-8. Overall percent correct of multiple choice programming questions for each condition, grouped by programming experience.

6.4.3 Performance Times

As shown in Table 6-12 all subjects completed the skyscraper charting task, P1, with the exception of three subjects in the passive training condition. Because of the missing data from the three subjects, the table presents mid-mean times for each condition, not mean times. To compute the mid-mean, those who did not complete the task were given the maximum allotted time (in this case 2700 seconds) and then the upper and lower quartiles for each condition were dropped. Means, confidence levels, and errors, were then calculated based on this filtered data. Unfortunately, there are no known inferencing techniques for the mid-mean. Therefore a weak nonparametric test, the Kruskal-Wallis one-way ANOVA of the median times, was used to compared differences between conditions over the entire data set. No statistically significant differences were found, which is not surprising considering the high variability in the times for all conditions.

A much smaller percentage of subjects successfully completed the skyscraper coloring task, P2, as shown in Table 6-13. Again, because of missing data—in this case from 13 subjects—the table presents mid-mean times for each condition. A Kruskal-Wallis test on the medians for the unfiltered dataset found no statistically significant differences.

Table 6-12. Results from performance task P1 grouped by condition. Number Completed is the count of subjects who successfully completed the task in the allotted 45 minutes (2700 seconds.) For each condition the table lists the mid-mean time in seconds, the 95% confidence level, standard error, and standard deviation.

	No Training	Passive Training	Active Training	Overall
Number Completed	10	7	10	27
Mid-Mean	1440	1842	995	1426
Conf. Level	421	762	740	
Std. Err.	152	274	266	
Std. Dev.	339	614	596	

Logs files from performance tasks P1 and P2 were examined for various signature phenomena between conditions, but none were found. The first two expression evaluations made by subjects following the introduction to the Language window and other Chart 'n' Art's disclosure mechanisms (events 12 and 13) showed no characteristic differences. Furthermore, an ANOVA of the number of expressions evaluated per subject as well as a comparison of the delay before a subject evaluated his or her first expression after event 13, revealed no statistically significant differences between conditions.

Table 6-13. Results from performance task P2 grouped by condition. Number Completed is the count of subjects who successfully completed the task in the allotted 10 minutes (600 seconds). For each condition the table lists the mid-mean time in seconds, the 95% confidence level, standard error, and standard deviation.

	No Training	Passive Training	Active Training	Overall
Number Completed	5	7	5	17
Mid-Mean	458	318	474	417
Conf. Level	232	183	163	
Std. Err.	84	66	59	
Std. Dev.	187	147	131	

6.4.4 Final Programming Task

Table 6-14 lists the mean programming language understanding scores for the final programming task, P3, in which subjects typed expressions from scratch without help from disclosures. The only significant difference in a pair-wise comparison of mean scores was the 1.05 point gap between the overall passive and active training groups ($F_{2,27} = 3.547$, $p < .0432$). When the scores were grouped by programming background, only the 1.36 point difference between P_L and A_L was found to be significant ($F_{2,27} = 5.283$, $p < .0198$).

Table 6-14. Degree of programming language understanding demonstrated in the final program that subjects wrote from scratch without help from disclosures. The understanding score is the mean understanding per condition as rated by judges on a scale from 1 to 6. For each condition the table lists mean score, the 95% confidence level, standard error, and standard deviation.

	No Training	Passive Training	Active Training	Overall
Overall Mean	3.00	2.40	3.45	2.95
Conf. Level	0.75	0.67	0.43	
Std. Err.	0.33	0.30	0.19	
Std. Dev.	1.05	0.94	0.60	
Low Exper. Mean	2.70	2.00	3.36	2.77
Conf. Level	1.21	0.76	0.51	
Std. Err.	0.44	0.27	0.21	
Std. Dev.	0.98	0.61	0.56	
High Exper. Mean	3.30	2.80	3.67	3.19
Conf. Level	1.43	1.36	1.90	
Std. Err.	0.52	0.49	0.44	
Std. Dev.	1.15	1.10	0.76	

A qualitative analysis of the final programs was conducted to examine patterns of programming language use that were independent of conditions. The programs showed a limited variety of solutions to the task of drawing and coloring the skyscrapers. All but four subjects used the method suggested by the hints provided in the earlier tasks. This technique involved generating 30 shapes and then sizing, positioning, and coloring them using various mapping expressions. Three subjects apparently intended to provide several lines of SK8Script for each of the 30 rectangles in order to specify properties for each shape such as position, size, and color. The remaining subject attempted to use a single repeat loop to create a new shape with the appropriate size, position, and color within each iteration of the loop. None of the programs would have quite executed correctly, but all consisted of a reasonable sequence of operations that looked like SK8Script. Although, according to the judges, the closeness to correct SK8Script varied considerably between subjects, it is worth noting a few common characteristics of subjects' code:

- 21 of the 30 subjects (11 low experience, 10 high experience) used forms which bore a strong resemblance to the SK8Script one-to-many mapping expression.
- 25 subjects (14 low, 11 high) wrote some semblance of a many-to-many mapping expression.
- 3 subjects (1 low, 2 high) used a repeat loop with one or more nested expressions for the loop's body.
- 18 subjects (9 low, 9 high) had some semblance of a test filter within a selection expression.

6.5 Discussion

The key results in Table 6-6 highlight interesting differences between conditions in the summative assessment. The fact that overall scores in the short-answer and multiple-choice test were significantly higher in the active training condition than the no-training condition suggests that, with encouragement to actively explore disclosed expressions, self-disclosing systems can help teach users elements of an end-user programming language. Let us examine the results in more detail, especially in light of the three main questions this experiment was designed to help answer.

6.5.1 Incidental Learning

The passive training group was intended to simulate incidental learning conditions in that during the training phase these subjects were not explicitly told to attend to disclosures nor could they edit or evaluate disclosures appearing in the Language window. However, the differences in the results between the passive training condition and the no training condition provide little evidence that incidental learning from disclosures actually occurred. The only statistically significant difference between these conditions resulted from the t-test of P_H and N_H scores on the short-answer test. An explanation for this difference is offered in the Section 6.5.3. Two other results hinted at the possibility of differences between the passive and no training groups: Odds were computed to be 9:1 that there was a difference (a) in overall short-answer scores and (b) in multiple-choice scores for syntax questions.

Given these results one might, despite the lack of statistical significance, employ the odds-ratio data to inform purchasing decisions. For instance, if a designer could choose between a standard drawing package and one which featured disclosure mechanisms, it would be helpful for the designer to know that the odds were 9:1 that he or she would incidentally learning about a potentially useful programming language while using the latter package. However, the lack of signature phenomena differentiating the passive and no training groups suggests that even this might be an overly optimistic perspective on incidental learning. On the other hand, perhaps these signature effects might have surfaced had users been exposed to disclosures for longer than the one-hour training period. Further study is clearly needed to help settle the incidental learning issue.

6.5.2 Composition

Quantitative gains in productive programming knowledge were difficult to assess in the Chart 'n' Art study. For one, timing results from the performance tasks P1 and P2 were inconclusive. Significantly worse times by the no training condition might have indicated that those exposed to disclosures could more quickly form SK8Script expressions, but this was not the case. It is possible that the hour-long training time was simply too short for subjects to acquire enough SK8Script knowledge to write expressions on their own. Another confounding issue was that during performance tasks P1 and P2, subjects almost always favored editing and evaluating disclosures rather than typing an expression from scratch. This behavior made it difficult to determine when subjects were actually producing code and when they were simply copying it.

In the final task in which subjects were forced to write code from scratch, characteristics of the resulting programs provide some qualitative insight into the composition question. Note that by the time subjects participated in this final production task all of them, including those in the passive and no training groups, had had the chance to see and use disclosures. For this reason, what is more interesting than differences between conditions (which were at best questionable) are the patterns of language use by all subjects. After exposure to mainly isolated single-line disclosures, all subjects regardless of programming experience, managed to devise a meaningful sequence of SK8Script-like expressions. Subjects' understanding of the concept of nested expressions is less clear. Although as many as 18 subjects had some notion of a selection expression with an embedded test filter (e.g. "`every item whose fillColor is Red`"), it is quite possible that subjects thought of such expressions as a whole rather than as a test nested within another expression.

More promising is the fact that the majority of subjects, with both high and low programming background, recognized the need for and were able to generate the semblance of SK8Script's mapping expressions. As argued in Chapter 2, these expressions combined with only a few other primitives can allow designers to accurately and efficiently create a wide variety of custom information graphics. The fact that many of these expressions contained vocabulary and syntactic errors which would have prevented them from executing properly is perhaps not as much of a concern as it would at first seem. Many of these errors may never have occurred were subjects given access to disclosed expressions to use as templates. Further study is needed to better understand the ways in which disclosures are similar to structured editors (Goldenson & Wang, 1991) in facilitating the production of accurate expressions.

6.5.3 Programming Experience Influence

Results from the summative assessment indicate that programming background can affect one's ability to learn from disclosures. A_L subjects tended to score better than N_L subjects on both the short-answer and multiple-choice tests. Meanwhile, A_H subjects did not outperform N_H subjects. The implication is that users with low programming experience who are encouraged to actively explore disclosed expressions are somehow able to learn better than users with a stronger background in programming who are similarly encouraged. One possible explanation for this phenomenon is that low-experience and high-experience users approach learning from disclosures with very different attitudes. Low-experience users may welcome a new way to learn about programming, a topic that had so far eluded them. High-experience users, on the other hand, may already have considerable skill in learning from traditional instructional resources, and are therefore less interested in attending to new forms of learning opportunities.

As mentioned earlier, there was another significant difference related to programming experience that was evident between the passive training group and the active training group in the short-answer test. P_H subjects tended to score better than N_H subjects, but P_L subjects did not outperform N_L subjects. The implication is that high programming experience users are able to learn better *incidentally* than users with a weaker background in programming. One interpretation is that those with more programming background are "primed" to recognize disclosed expressions and will therefore be more likely to be interested in and attend to them on their own. A related explanation is that only those with more programming background have the cognitive structures necessary to make sense of disclosed expressions. To use Vygotskian terms, disclosures are within these users' "zone of proximal development" (Vygotsky, 1978).

Note that the first phenomenon implied that low programming experience users can better take advantage of disclosures under active training conditions, while the second phenomenon gives the advantage to high experience users under incidental learning conditions. Although this interaction between condition and experience was not shown to be significant in either the short-answer or multiple-choice test results, from an odds-ratio perspective the chances were better than 12:1 that it existed in the short-answer test and better than 5:1 that it existed in the multiple-choice test. Such an interaction between condition and experience, if it does indeed occur, would be curious, but not unexplainable. Since the second phenomenon deals with subjects learning incidentally without being explicitly told to attend to disclosures, the attitudinal issues raised in the first phenomenon may not apply. In other words, high experience subjects may be less resistant to learning from disclosures when they are not being forced to do so. They may, in fact, consider themselves clever for having noticed the disclosures and attend to them all the more.

6.5.4 Overall Interpretation

There are a number of ways to interpret the overall results. In fact, reviewers of this document disagree amongst themselves on the conclusions to be drawn. Generally, there are two main interpretations: a negative one suggesting that self-disclosure has yet to prove itself as a viable means for learning programming, and a positive one suggesting that self-disclosure shows promise as an effective vehicle for helping designers become oriented to a programming language. This chapter concludes by summarizing both of these perspectives.

The negative interpretation. The fact that subjects in the active training condition performed significantly better than those in the other two conditions is no surprise, since they were forced to attend to disclosures and make use of them as much as possible. This would be expected regardless of the instructional method used as long as the other subjects did not receive the instruction or were not required to attend to it. For example, the experiment could have alternatively offered active training subjects on-line help messages describing programming techniques. The most important advantage self-disclosure could have over these other approaches is that users might learn incidentally from exposure to programming language feedback. However, the differences in performance between passive and no training groups failed to show that users could actually learn in this way. The few subjects who did seem to learn some SK8Script in the passive training condition were those with higher programming experience, but it is likely that these subjects could just as easily have picked up elements of the language in a number of other ways. For those with lower programming experience, disclosures were not effective for learning programming by passive observation. Perhaps disclosures are unsuitable for those with limited programming experience who might see the disclosures as little more than a constant barrage of random text.

The positive interpretation. The fact that subjects were able to learn anything at all about programming in SK8Script simply from observing language disclosures is a remarkable result. The experiment demonstrates that the unstructured but situated feedback of self-disclosure can be a means for users to become familiar with a programming language. The lack of evidence for incidental learning only suggests that users perhaps need to be somewhat attentive to disclosures to benefit from them. On the other hand, just because only the active training group showed significant learning effects does not mean that users need to constantly fixate on disclosures. It is possible that occasional attention is sufficient. Furthermore, the small but not significantly better scores by subjects in the passive training condition suggests the possibility that the cumulative long term benefits of self-disclosure could be meaningful. Finally, the fact that those with higher programming experience seemed to learn best from disclosures should not be seen as evidence that self-disclosure is inappropriate for beginning programmers. Rather, it simply indicates that self-disclosure might work best for these users in combination with a more structured programming language curriculum.

7. Conclusion

The research presented in the preceding chapters makes a case for self-disclosing design tools as an incremental approach toward end-user programming. This chapter briefly reviews the arguments and evidence, identifies some limitations and future research, and finally highlights the contributions of this work.

7.1 Review

Chapter 2 presented four case studies of designers creating custom information graphics with general-purpose drawing tools. These designers showed remarkable resistance to more accurate and more automatic aids such as spreadsheets or charting software. It was hypothesized that these tools were rejected because of their perceived rigidity and because of designers' lack of familiarity with the interfaces to such tools. Programmable drawing tools were presented as a potential solution for each case study, because their language-based computational support could be tightly integrated with a familiar direct manipulation interface that imposed few constraints on the design space. However, the designers' unique work environments and frustrations with existing educational materials indicated that teaching them about using an embedded programming language would be a challenge.

The following chapter proposed self-disclosure as a technique for gently introducing designers to programming. Self-disclosure was defined as the act of the computer volunteering hidden or non-obvious information about its status or functionality. It was argued that a self-disclosing programmable tool could offer designers situated, incremental language learning opportunities that addressed many of the concerns about educational materials identified in the case studies. This chapter also proposed a set of educational dimensions for characterizing self-disclosing systems and recommended points along these dimensions that were most likely to specify pedagogically effective systems.

Chart 'n' Art was then presented as a prototype self-disclosing tool that comes close to matching the profile of an pedagogically effective system. Chapter 4 described Chart 'n' Art's four main disclosure techniques in detail, and how the system attempts to immerse users in its programming language by taking almost every opportunity to reveal short programming examples. A comparison between Chart 'n' Art and other computer-based tools for graphic design and programming language education suggested that its learning-while-doing approach was unique. It appears that no other graphic design tool had as much emphasis on education, and no other educational tool had as much emphasis on graphic design.

Chapter 6 provided experimental evidence indicating that not only do Chart 'n' Art's disclosure mechanisms facilitate the use of its language in combination with direct manipulation activity, but that users can gain programming language expertise by attending to its disclosures. Limited evidence from the summative assessment of Chart 'n' Art suggested that low experience programmers benefited most when actively attending to disclosures, while high experience programmers could become oriented to Chart 'n' Art's particular language incidentally.

7.2 Limitations

As conceded earlier, Chart 'n' Art represents only a small region of points within the multidimensional design space for pedagogical self-disclosing tools. As such, Chart 'n' Art provides only a limited view of the potential role of self-disclosure in teaching programming to end-users. A broader picture is needed that, for example, encompasses the following additional points in the design space:

- ***High Adaptivity***. If a self-disclosing system could dynamically adjust its feedback according to an individual's ability, would the system have significantly greater educational value?

- ***Very High Explorability***. What if disclosures were part of an elaborate hypertext system that provided links to related examples, complete programs, or a language reference manual? What if there was unlimited undo?

- ***High Intrusiveness***. How much more memorable are disclosures that require explicit dismissal by the user? How would such disclosures be received by designers and what would be their impact on design flow?

- ***Low Relevance***. If disclosures were more random than in Chart 'n' Art, would users be able to make sense of them?

- ***High Scope***. Will users form a more complete understanding of a system's language if disclosures guarantee coverage over all vocabulary and syntax?

- ***High Structure***. Would it be more educational if disclosures could be ordered? If so, should the curriculum be patterned after traditional computer science classes?

The dimensions identified earlier are by no means exhaustive. It is likely that these dimensions will need to be refined and augmented to accurately describe the pedagogical characteristics of self-disclosing systems. For example, a potential candidate for a new dimension is "richness" which could refer to the variety of perspectives used to describe internal status or functionality. At one end of this spectrum would be systems such Apple's Script Editor which provide only literal, executable "recordings" of programs, and at the other end would be tools such as Redmiles' Explainer (Redmiles, 1993) that offer multiple views on example programs possibly including explanatory text, diagrams, or high-level descriptions of machine "transactions" (Mayer, 1979).

Even within the small region of the self-disclosure design space occupied by Chart 'n' Art, a number of important questions remain unanswered. As mentioned in the previous chapter, further study is needed to assess the incidental learning effects from self-disclosure and the impact this has on users of varying programming background. Additional unresolved issues include the following:

- ***Long Term Effects***. Studies of Chart 'n' Art have investigated learning over sessions totaling at most two or three hours. What are the long term educational benefits of disclosures? Will users learn to pay more and more attention or will their interest wane?

- ***Procedures and Persistence***. Chart 'n' Art does not disclose how to write procedures nor does it allow expressions to be saved to a file. How should the system support procedural composition and the ability to store programs for use in future sessions or to share with colleagues?

- ***Generality***. The language used by Chart 'n' Art to describe direct manipulation drawing actions is highly imperative. How feasible would it be to disclose a declarative representation of a drawing, specifying relations between objects and data? How applicable are Chart 'n' Art's disclosure mechanisms to other domains of programmable applications such as symbolic mathematics tools or simulation environments?

- ***Feasibility***. In the current implementation of Chart 'n' Art, although the drawing module is not required to produce the actual text of every disclosed expression and explanation, it nonetheless must construct canonical language structures for most disclosures. Obviously, this kind of burden limits the commercial feasibility of self-disclosure. How could developers render a tool self-disclosing without such invasive measures?

- ***Thoroughness***. Many useful programming topics such as recursion, procedural abstraction, and progressive refinement are not covered by Chart 'n' Art disclosures. How might Chart 'n' Art be used in conjunction with more traditional educational materials to provide a more thorough programming education?

7.3 Future Work

Further investigation is being considered to specifically address some of the above limitations. One proposal is to enable more of the self-disclosure design space to be explored by developing a modular, flexible disclosure architecture as shown in Figure 7-1. Under this model, a tool handles user interaction and broadcasts "aggregate events" describing manipulations to an artifact or construction. Aggregate events are similar to Apple Events only they can describe a hierarchy of actions including low level mouse movements and also high level operations such as the use of menu commands (Kosbie & Myers, 1993). Events are translated by a tool-specific disclosure generator into CLSs which describe the actions in terms of alternative expressions in a canonical language. These CLSs are then adapted for a particular curriculum by a disclosure filter that potentially draws from a user model to arrive at a set of CLSs to actually be disclosed. The disclosure presenter is responsible for translating the filtered CLSs into a specific language, generating explanations, and displaying the text in a separate window or by the cursor. Tightly connected to the presenter is a programming language interpreter in which users can type expressions from scratch or employ disclosures as templates. Evaluating these expressions causes aggregate events to be sent back to the tool to affect the current construction.

Using the above architecture, a variety of user modules could be tested to improve adaptivity without reimplementing the tool or a basic disclosure mechanism. Hypertext links to disclosures could be implemented by plugging in a more sophisticated disclosure presenter component. Disclosure generators could be substituted to achieve varying degrees of relevance, scope, and structure. Note that this modular architecture would also help address the feasibility issue by making the tool responsible for only broadcasting aggregate events, not the disclosures themselves.

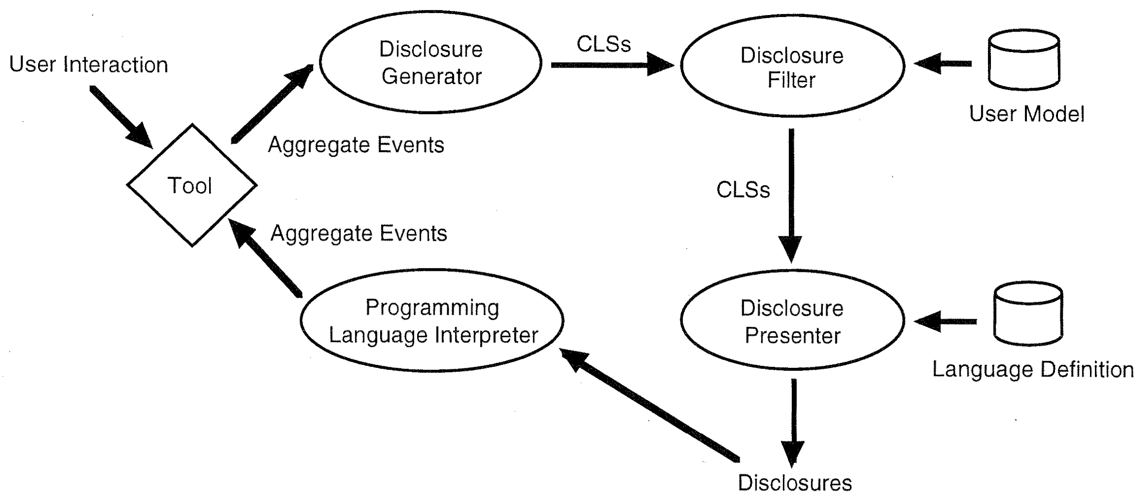


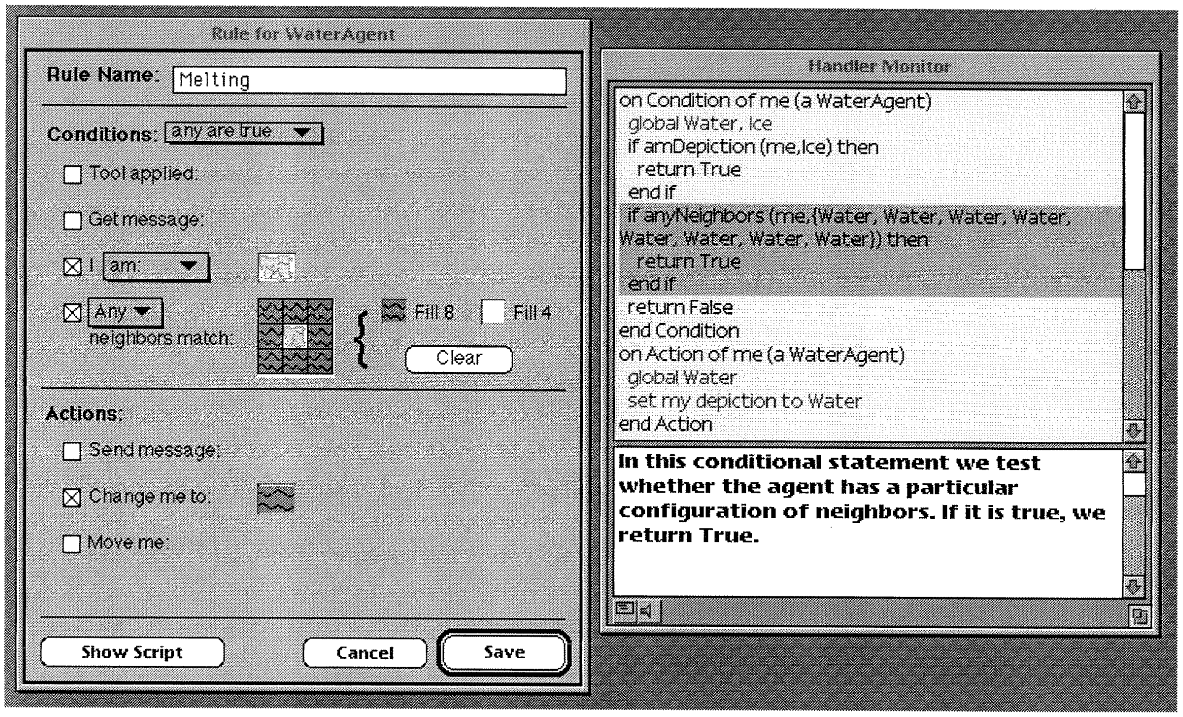
Figure 7-1. A modular language disclosure architecture that would facilitate experimentation with a variety of feedback mechanisms.

To address the issue of long-term effects, a longitudinal study has been proposed that would measure educational outcomes from a self-disclosing system over a period of several months. Rather than test Chart 'n' Art, the focus would be on a robust environment with an existing sophisticated user population: the Mac OS Finder. The Finder is the familiar desktop interface to the file system. Since it is a "recordable" application (see Section 5.2), a separate Script Editor window can be configured to disclose AppleScript as users manipulate files. Subjects would be provided with an extra monitor for displaying these disclosures without compromising subjects' normal screen real estate. Subjects would be recruited who might particularly benefit from AppleScript knowledge: users operating on large numbers of files or performing repetitive filing tasks. Although AppleScript "recordings" are missing many of the disclosure mechanisms in Chart 'n' Art (e.g. explanations, alternatives, selections labels, and mouse text), such a study could nonetheless reveal interesting learning patterns due to long-term exposure to a variety of programming examples.

Ongoing self-disclosure research could potentially address other questions posed by this report. For example, a screen saver is being developed for Chart 'n' Art which randomly permutes a user's construction (e.g. by creating, deleting, moving, and resizing shapes) and simultaneously discloses the relevant SK8Script expressions (DiGiano & Eisenberg, 1996). This more random mechanism allows the system to reveal programming techniques that it might not otherwise present to the user.

Another line of research has involved adding disclosures to the AgentSK8 simulation environment. AgentSK8, based on Agentsheets (Repenning & Sumner, 1995), is a tool that allows users to design, execute, and interact with simulations of a large variety of phenomena from physics to biology to sociology. Simulations consist of autonomous user-defined iconic "agents" whose behavior is specified by pairs of condition-action procedures written in a derivative of SK8Script called "AgenTalk." Users construct these procedures automatically when they manipulate a number of on-screen forms for indicating conditions and actions through check boxes, menus, and drag and drop operations. Disclosure was added to make this procedure construction process visible, so that users could learn AgenTalk programming in order to eventually define more complex behavior than is possible using the forms. To this end, the procedures are "monitored" in an adjacent window tightly linked to the forms as shown in Figure 7-2.

Changing a check box, menu selection, etc. causes corresponding changes in the procedures to be highlighted. Explanations for selected portions of the code can appear in a text pane beneath the procedures and can also be read out loud to the user by a voice synthesizer. More information on disclosures in AgentSK8 can be found in (DiGiano, 1996).



Rule Name: Melting

Conditions: any are true

☐ Tool applied:

☐ Get message:

☒ I am:

☒ Any neighbors match: ☒ Fill 8 ☐ Fill 4

Actions:

☐ Send message:

☒ Change me to:

☐ Move me:

```
on Condition of me (a WaterAgent)
  global Water, Ice
  if amDepiction (me,Ice) then
    return True
  end if
  if anyNeighbors (me,{Water, Water, Water, Water,
Water, Water, Water, Water}) then
    return True
  end if
  return False
end Condition
on Action of me (a WaterAgent)
  global Water
  set my depiction to Water
end Action
```

In this conditional statement we test whether the agent has a particular configuration of neighbors. If it is true, we return True.

Figure 7-2. Monitoring disclosures in AgentSK8. As users interact with the form on the left for specifying agent behavior, the code on the right is automatically updated and highlighted.

7.4 Contributions

This report provides a unique perspective on the language learnability problem of programmable applications. Rather than try to simplify or hide the language as many programming by demonstration systems have done, this work seeks to make the language more accessible. Rather than make a language more accessible by explicit teaching through textbooks or training classes, this work investigates ways in which learning can take place implicitly through self-disclosing tools. Chart ‘n’ Art, with its rich set of feedback mechanisms, is offered as a model for such pedagogical self-disclosing tools. It is the first programmable application that embeds language learning opportunities directly into the interface.

The self-disclosure approach is predicated on two key philosophies: (1) that end-users should be trusted with a full-fledged programming language that they can gradually master, perhaps over a number of years; and (2) that computer-based education can take on relatively benign and incremental forms which, given time, can nonetheless be valuable learning experiences. Self-disclosure is not intended for users who are necessarily in a hurry to learn, or for those expecting their education to be highly structured. Instead, the technique is for designers with more immediate goals than becoming “power-users” or end-user programmers. First and foremost, they want to create beautiful art, generate elegant information graphics, or develop insightful simulations. Self-disclosing design tools respect this priority and at the same time acknowledge the possibility that users might want to eventually advance their skills.

The evaluation of Chart 'n' Art suggests that its particular choice of echoing, identifying, and advertising techniques can be a means for at least some individuals to become oriented to an end-user language. However, in contrast to the initial hope that self-disclosure could be a means for lowering the threshold of programming for all, the summative assessment suggests that disclosures by themselves provide insufficient learning opportunities for those with limited programming background who are not actively attending to language feedback. This may be because of the particular disclosure mechanisms in Chart 'n' Art, but it may also be evidence of individual learning styles. Perhaps these users need some combination of more traditional instruction and unstructured disclosure feedback. Perhaps it is only a certain type of designer, those who become the local "gardeners" for instance, who have the curiosity and motivation to make use of disclosures to learn programming.

In conclusion, this research offers important lessons for developers of high-functionality computational design tools. For those with programmable tools, this work presents examples of how they might go about embedding self-disclosure mechanisms into their systems as a way to increase the accessibility and possibly the learnability of these systems' languages. For those with design tools that are not currently programmable, this research gives reason to reconsider the idea of adding end-user programming facilities. Developers of these tools may have initially rejected such facilities because it seemed their users did not want to program. It is possible, however, that the real issue was that their users were concerned about the time-consuming additional learning burden of becoming oriented to a formal language for their design activity. By embedding learning opportunities within familiar software, self-disclosure offers a means of learning-while-doing that could possibly make programming a more practical design tool for those who have not considered using it before.

References

Adobe Systems Inc. (1996a). *Illustrator*. San Jose, CA.

Adobe Systems Inc. (1996b). *PageMaker*. San Jose, CA.

Adobe Systems Inc. (1996c). *Photoshop*. San Jose, CA.

Aitchison, J. (1985). Predestinate grooves: is there a preordained language "program"? In V. P. Clark, P. A. Eschholz, & A. F. Rosa (Eds.), *Language: Introductory Readings*, (4 ed.,). New York: St. Martin's Press.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: lessons learned. *Journal of Learning Sciences*, 4(2), 167-207.

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review*, 86(2), 125-140.

Apple Computer Inc. (1993). *AppleScript Language Guide*. Reading, MA: Addison-Wesley.

Apple Computer Inc. (1995). *Inside Macintosh: Interapplication Communication*. Reading, MA: Addison-Wesley.

Apple Computer Inc. (1996a). *HyperCard*. Cupertino, CA.

Apple Computer Inc. (1996b). *SK8 User's Guide*. Cupertino, CA.

AutoDesk Inc. (1996). *AutoCAD*. San Rafael, CA.

Bell, B., Citrin, W., Lewis, C., Rieman, J., Weaver, R., Wilde, N., & Zorn, B. (1994). Using the programming walkthrough to aid in programming language design. *Journal of Software Practice and Experience*, 24(1), 1-25.

Berlin, L. M., & Jeffries, R. (1992). Consultants and apprentices: observations about learning and collaborative problem solving. In *CSCW '92 Proceedings*, (pp. 130-137). New York: ACM Press.

Bolt, R. A. (1984). *Future interfaces*. Belmont, CA: Lifetime Learning Publications.

Boulay, B. d., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 237-250.

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

Burton, R. R., & Brown, J. S. (1976). A tutoring and student modeling paradigm for gaming environments. In *Proceedings of the Symposium on Computer Science and Education*. Anaheim, CA.

Carroll, J. M., & Rosson, M. B. (1987). Paradox of the active user. In J. M. Carroll (Ed.) *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, (pp. 80-111). Cambridge, MA: The MIT Press.

Claris Corp. (1996). *MacDraw*. Cupertino, CA.

Collins, A. M., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.) *Knowing, Learning, and Instruction*, (pp. 453-494). Hillsdale, NJ: Lawrence Erlbaum Associates.

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. New York: Harper & Row.

Cypher, A. (1993). Eager: programming repetitive tasks from examples. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 405-413). Cambridge, MA: The MIT Press.

DeltaPoint Inc. (1996). *DeltaGraph Pro*. Monterey, CA.

Derlega, V. J., & Berg, J. H. (1987). *Self-Disclosure*. New York: Plenum Press.

DiGiano, C. (1996). A vision of highly-learnable end-user programming languages. In A. Repenning, R. Hubscher, & C. Lewis (Eds.), *Child's Play '96 Position Papers*.

DiGiano, C. (in preparation). *Chart 'n' Art User Manual* (CU-CS-#): Department of Computer Science, University of Colorado at Boulder.

DiGiano, C., & Eisenberg, M. (1995). Self-disclosing design tools: a gentle introduction to end-user programming. In G. Olson & S. Schuon (Eds.), *Proceedings of DIS '95 Symposium on Designing Interaction Systems*, (pp. 189-197). New York: ACM Press.

DiGiano, C., & Eisenberg, M. (1996). Designing pedagogical screen savers. In J. D. Mackinlay (Ed.) *CHI '96 Conference Companion*, (pp. 185-186). New York: ACM Press.

Eisenberg, M. (1995). Programmable applications: interpreter meets interface. *SIGCHI Bulletin*, 27(2), 68-93.

Eisenberg, M., & Fischer, G. (1994). Programmable design environments: integrating end-user programming with domain-oriented assistance. In B. Adelson, S. Dumais, & J. Olson (Eds.), *CHI '94 Proceedings*, (pp. 431-437). New York: ACM Press.

Elley, W. B. (1989). Vocabulary acquisition from listening to stories. *Reading Research Quarterly*, 24(2), 175-187.

Elsom-Cook, M. (1990). Guided discovery tutoring. In M. Elsom-Cook (Ed.) *Guided Discovery Tutoring: A framework for ICAI research*, (pp. 3-23). London: Paul Chapman Publishing.

Fischer, G. (1987). A critic for LISP. In J. McDermott (Ed.) *The 10th International Joint Conference on Artificial Intelligence*, (pp. 177-184). Los Altos, CA: Morgan Kaufmann Publishers.

Fischer, G. (1991). Supporting learning on demand with design environments. In L. Birnbaum (Ed.) *Proceedings of the International Conference on the Learning Sciences 1991*, (pp. 165-172). Charlottesville, VA: Association for the Advancement of Computing in Education.

Fischer, G. (1993). Conceptual frameworks and computational environments in support of learning on demand. In *NATO Advanced Workshop on The Design of Computational Media to Support Exploratory Learning*. Asilomar.

Fischer, G., & Mørch, A. (1988). CRACK: a critiquing approach to cooperative kitchen design. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, (pp. 176-185). New York: ACM Press.

Fischer, G., & Nakakoji, K. (1991). Making design objects relevant to the task at hand. In *Proceedings of AAAI-91, Ninth National Conference on Artificial Intelligence*, (pp. 67-73). Cambridge, MA: AAAI Press/The MIT Press.

Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., & Sumner, T. (1993). Embedding computer-based critics in the contexts of design. In *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, (pp. 157-164).

Fischer, G., & Stevens, C. (1987). Volunteering information—enhancing the communication capabilities of knowledge-based systems. In H. J. Bullinger & B. Shackel (Eds.), *Proceedings of INTERACT '87, 2nd IFIP Conference on Human-Computer Interaction*, (pp. 965-971). Amsterdam: North-Holland.

Fromkin, V., Krashen, S., Curtiss, S., Rigler, D., & Rigler, M. (1985). The development of language in Genie: a case of language acquisition beyond the critical period. In V. Clark, P. Eschholz, & A. Rosa (Eds.), *Language Introductory Readings*, (pp. 112-130). New York: St. Martin's Press.

Gantt, M., & Nardi, B. A. (1992). Gardeners and gurus: patterns of cooperation among CAD users. In *CHI '92 Proceedings*, (pp. 107-117). New York: ACM.

Glasgow, D. (1994). *Information Illustration*. Reading, MA: Addison-Wesley.

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

Goldenson, D. R., & Wang, B. J. (1991). Use of structure editing tools by novice programmers. In J. Koenenmann-Belliveau, T. G. Moher, & S. P. Robertson (Eds.), *Empirical Studies of Programmers: Fourth Workshop*, (pp. 99-120).

Halbert, D. C. (1993). SmallStar: programming by demonstration in the desktop metaphor. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 103-123). Cambridge, MA: The MIT Press.

Heydon, A., & Nelson, G. (1994). *The Juno-2 Constraint-Based Drawing Editor* (Technical Report 131a): Systems Research Center, Digital Equipment.

Houghton, R. C. (1984). Online help systems: a conspectus. In *Communications of the ACM*, (Vol. 27, pp. 126-133).

Jackiw, R. N., & Finzer, W. F. (1993). The geometer's sketchpad: programming by geometry. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 293-307). Cambridge, MA: The MIT Press.

Joe, A. (1995). Text-based tasks and incidental vocabulary learning. *Second Language Research*, 11(2), 149-157.

Kosbie, D. S., & Myers, B. A. (1993). A system-wide macro facility based on aggregate events: a proposal. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 433-444). Cambridge, MA: The MIT Press.

Kurlander, D. (1993). Chimera: example-based graphical editing. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 271-291). Cambridge, MA: The MIT Press.

Kurlander, D., & Feiner, S. (1993). A history of editable graphical histories. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 405-413). Cambridge, MA: The MIT Press.

Landauer, T. K. (in press). Behavioral research methods in human-computer interaction. In M. Helander, T. K. Landauer, & P. Prabhu (Eds.), *Handbook of Human-Computer Interaction*, (2nd ed.). New York: North Holland.

Lewis, C. (1988). Why and how to learn why: analysis-based generalization of procedures. In *Cognitive Science*, (Vol. 12, pp. 211-256).

Lewis, C., Hair, D. C., & Schoenberg, V. (1989). Generalization, consistency, and control. In *CHI '89 Proceedings*, (pp. 1-5). New York: ACM Press.

Lewis, C., & Olson, G. M. (1987). Can principles of cognition lower the barriers to programming? In *Empirical Studies of Programmers: Second Workshop*, (pp. 248-263).

Lieberman, H. (1993). Mondrian: teachable graphical editor. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 341-359). Cambridge, MA: The MIT Press.

Mackay, W. E. (1991). Triggers and barriers to customizing software. In *CHI '91 Proceedings*, (pp. 153-160). New York: ACM.

Macromedia Inc. (1996a). *Director*. San Francisco, CA.

Macromedia Inc. (1996b). *FreeHand*. San Francisco, CA.

Mayer, R. E. (1979). A psychology of learning BASIC. *Communications of the ACM*, 22(11), 589-593.

Microsoft Corp. (1995). *Excel 5.0 Visual Basic User Guide*. Redmond, WA: Microsoft Corp.

Microsoft Corp. (1996a). *Excel*. Redmond, WA.

Microsoft Corp. (1996b). *Visual BASIC*. Redmond, WA.

Microsoft Corp. (1996c). *Word.* Redmond, WA.

Miller, M. L. (1982). A structured planning and debugging environment for elementary programming. In D. H. Sleeman & J. S. Brown (Eds.), *Intelligent Tutoring Systems*, (pp. 119-135). London - New York: Academic Press.

Modugno, F., Corbett, A. T., & Myers, B. A. (1995). Evaluating program representation in a demonstrational visual shell. In I. Katz, R. Mack, & L. Marks (Eds.), *CHI '95 Conference Companion*, (pp. 234-235). New York: ACM Press.

Myers, B. A., Goldstein, J., & Goldberg, M. A. (1994). Creating charts by demonstration. In B. Adelson, S. Dumais, & J. Olson (Eds.), *CHI '94 Proceedings*, (pp. 106-111). New York: ACM Press.

Nakakoji, K., Reeves, B. N., Aoki, A., Suzuki, H., & Mizushima, K. (1995). eMMaC: knowledge-based color critiquing support for novice multimedia authors. In P. Zellweger (Ed.) *ACM Multimedia '95 Proceedings*, (pp. 467-476). New York: ACM Press.

Nakakoji, K., Sumner, T., & Harstad, B. (1994). Perspective-based critiquing: helping designers cope with conflicts among design intentions. In J. S. Gero & F. Sudweeks (Eds.), *Proceedings of Artificial Intelligence in Design '94*, (pp. 449-466). Amsterdam: Kluwer Academic Publishers.

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

Neuman, S. B., & Koskinen, P. (1992). Captioned television as comprehensive input: Effects of incidental word learning from context for language. *Reading Research Quarterly*, 27(1), 95-106.

Newton, J. (1995). Task-based interaction and incidental vocabulary learning: a case study. *Second Language Research*, 11(2), 158-177.

Norman, D. A., & Spohrer, J. C. (1996). Learner-Centered Education. *Communications of the ACM*, 39(4), 24-27.

Olson, J. S., Olson, G. M., Storrosten, M., & Carter, M. (1992). How a group-editor changes the character of a design meeting as well as its outcome. In *CSCW '92 Proceedings*, (pp. 91-98).

Owen, D. (1986). Answers first, then questions. In S. W. D. D.A. Norman (Ed.) *User Centered System Design, New Perspectives on Human-Computer Interaction*, (pp. 361-375). Hillsdale, NJ: Lawrence Erlbaum Associates.

Papert, S. (1991). Situating constructionism. In I. Harel & S. Papert (Eds.), *Constructionism*, (pp. 1-11). Norwood, NJ: Ablex Publishing.

Papert, S. (1993). *Mindstorms: Children, Computers, and Powerful Ideas.* (2nd ed.). New York: HarperCollins Publishers.

Piernot, P. P., & Yvon, M. P. (1993). The AIDE project: an application-independent demonstrational environment. In A. Cypher (Ed.) *Watch What I Do: Programming by Demonstration*, (pp. 383-401). Cambridge, MA: The MIT Press.

Pinker, S. (1979). Formal models of language learning. *Cognition*, 7, 217-283.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 39, 240-272.

Quark Inc. (1996). *QuarkXPress*. Denver, CO.

Redmiles, D. F. (1993). Reducing the variability of programmers' performance through explained examples. In *INTERCHI '93 Proceedings*, (pp. 67-73). New York: ACM.

Repenning, A., & Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3), 17-25.

Resnick, L. B. (1989). Introduction. In *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, (pp. 1-24). Hillsdale, NJ: Lawrence Erlbaum Associates.

Richards, I. A., & Gibson, C. M. (1945). *English through pictures*. New York: Pocket Books.

Rieman, J. (in press). A field study of exploratory learning strategies. *ACM Transactions on Computer-Human Interaction.*

Rosson, M. B., Carroll, J. M., & Bellamy, R. K. E. (1990). Smalltalk scaffolding: a case study of minimalist instruction. In J. C. Chew & J. Whiteside (Eds.), *CHI '90 Proceedings*, (pp. 423-429). New York: ACM.

Roth, S. F., Kolojejchick, J., Mattis, J., & Goldstein, J. (1994). Iteractive graphic design using automatic presentation knowledge. In B. Adelson, S. Dumais, & J. Olson (Eds.), *CHI '94 Proceedings*, (pp. 112-117). New York: ACM Press.

Selker, T. (1994). COACH: a teaching agent that learns. *Communications of the ACM*, 37(7), 92-99.

Sheppard, S. B., Curtis, B., Milliman, P., & Love, T. (1979). Modern Coding Practices and Programmer Performance. *IEEE Computer*, 12(12).

Snow, C. E. (1986). Conversations with children. In P. Fletcher & M. Garman (Eds.), *Language Acquisition: Studies in First Language Development*, (pp. 69-89). Cambridge, England: Cambridge University Press.

Spohrer, J., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy pascal programs. *Human-Computer Interaction*, 1(2).

Stallman, R. M. (1981). EMACS, the extensible, customizable, self-documenting display editor. *ACM SIGOA Newsletter*, 2(1/2), 147-156.

Totterdell, P. A., Norman, M. A., & Browne, D. P. (1987). Levels of adaptivity in interface design. In H. J. Bullinger & B. Shackel (Eds.), *Proceedings of INTERACT '87, 2nd IFIP Conference on Human-Computer Interaction*, (pp. 715-722). Amsterdam: North-Holland.

Tufte, E. R. (1983). *The Visual Display of Quantitative Information*. Cheshire, CT: Graphics Press.

UserLand Software Inc. (1996). *Frontier*. San Francisco, CA.

Venolia, D. (1994). T-Cube: a fast, self-disclosing pen-based alphabet. In B. Adelson, S. Dumais, & J. Olson (Eds.), *CHI '94*, (pp. 265-70). New York: ACM Press.

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

Wiedenbeck, S. (1989). Learning iteration and recursion from examples. *Int. J. of Man-Machine Studies*, 30, 1-22.

Wolfram Research Inc. (1996). *Mathematica*. Champaign, IL.

Appendix A
Summative Assessment Phone Interview Questions

How to answer a call from a prospective user study participant

Tell them what it's about

1. **You will use a new kind of drawing software being developed at CU to make some diagrams.**
2. **The experiment will last approximately three hours including a ten minute break.**
3. **Your screen will be videotaped, but not you.**
4. **The study runs from April 27 - June 1 in 3 hour slots. Early slots may fill up quickly due to people leaving at the end of the semester.**
5. **You will have to come to the Engineering Center on campus. The exact location to be determined.**
6. **Pay is $45.**

Ask them eligibility questions
If they do not qualify, thank them, and hang up.

1. Are you a University of Colorado student?

If they are either an undergraduate or graduate, proceed to question 2. Otherwise, they do not qualify.

2. Have you used a Macintosh drawing program in the last 4 years?

Examples of Mac Drawing programs:
MacDraw, SuperPaint, Canvas, Freehand, Illustrator, ClarisDraw, the drawing component of ClarisWorks, the built-in drawing tools in Microsoft Word.
If they have **any** level of experience, proceed to question 3. Otherwise, they do not qualify.

3. Have you ever used the programming language SK8Script—a new language being developed by Apple.

If they have **never** used SK8Script (very likely), proceed to question 4. Otherwise, they do not qualify.

4. Do you know Chris DiGiano or his research?

If they **do not** know about me or my work **proceed to the information questions**. Otherwise, they do not qualify.

Informational Questions for Participant #

The following information will be used to contact you for scheduling purposes.

1. Name: ____

2. Daytime phone: ____

3. Evening phone: ____

4. Email address (if applicable): ________________________________

Please tell us about your level of experience with various Macintosh drawing programs. By drawing programs we mean software which lets you draw basic shapes like rectangles and ovals and then allows you to move these around and change their appearance.

5. Use the following numerical rating scale to indicate your drawing program experience.

(F.: You might have to remind them several times about this rating scale)

1	2	3	4
very experienced	somewhat experienced	not very experienced	not at all experienced

Experience level (1-4)				*Drawing Program*
1	2	3	4	Canvas
1	2	3	4	ClarisDraw
1	2	3	4	Freehand
1	2	3	4	Illustrator
1	2	3	4	MacDraw
1	2	3	4	SuperPaint
1	2	3	4	the built-in drawing tools in Microsoft Word
1	2	3	4	the drawing component of ClarisWorks
1	2	3	4	Other: ______________________
1	2	3	4	Other: ______________________
1	2	3	4	Other: ______________________

(OVER)

Please tell us about your programming background. By programming we mean specifying an instruction or series of instructions to a computer typically by typing. Programming languages fall along a spectrum of complexity from macro languages found in spreadsheets to languages used by professional programmers such as C++.

6. Use the following numerical rating scale to indicate your programming experience in various languages.

(F.: You might have to remind them several times about this rating scale)

1	2	3	4
very experienced	somewhat experienced	not very experienced	not at all experienced

Experience level (1-4)				*Language*
1	2	3	4	Ada
1	2	3	4	Algol
1	2	3	4	AppleScript
1	2	3	4	Assembler
1	2	3	4	AutoLisp
1	2	3	4	BASIC
1	2	3	4	C
1	2	3	4	C++
1	2	3	4	COBOL
1	2	3	4	Fortran
1	2	3	4	HyperTalk
1	2	3	4	Java
1	2	3	4	Lingo
1	2	3	4	Lisp
1	2	3	4	Mathematica
1	2	3	4	Newton Script
1	2	3	4	Pascal
1	2	3	4	Perl
1	2	3	4	PL1
1	2	3	4	Prolog
1	2	3	4	Scheme
1	2	3	4	SmallTalk
1	2	3	4	VisualBASIC
1	2	3	4	Other: ________________
1	2	3	4	Other: ________________
1	2	3	4	Other: ________________

7. If you have ever programmed, how many years have you been programming?

1 - 3 years   3 -5 years   5-10 years   over 10 years

Participants can expect me to call them within a week or so, to schedule them.

I will call you, F., daily to get the latest information from people who have called. Don't bother to send me email.

Appendix B
Summative Assessment Demographic Questionnaire

Chart 'n' Art User Study

Entrance Questionnaire

Please answer the following questions by either circling one or more of the answers provided or by filling in a blank.

1. What is your sex? Female Male

2. Approximately how old are you?

<15 15-25 25-35 35-45 45-55 55-65 >65

3. What is your occupation?

____

4. If you are a student, what year are you in school?

____

5. If you are not a student, who is your employer?

____

6. Do you consider yourself to be a graphic designer or graphic artist?

Yes No

7. If you answered Yes to the previous question, how would you rate your ability to create graphics to be used in presentations compared to other graphic designers or graphic artists?

below average average above average

8. If you answered No to question 6, how would you rate your ability to create graphics to be used in presentations?

below average average above average

9. How would you rate your experience with software for making charts and graphs such as bar charts, pie charts, or x-y plots?

very experienced somewhat experienced not very experienced not at all experienced

10. A program is a way to specify an instruction or series of instructions to a computer typically by typing. Programming languages fall along a spectrum of complexity from macro languages found in spreadsheets to languages used by professional programmers such as C++.

Have you ever written a program for the Macintosh in particular?

Yes No

11. If you answered Yes to the previous question, have you ever written a program which draws graphics on the Macintosh, that is produces something other than text on the screen?

Yes No

12. Are you currently using programming languages in your work?

Yes No

13. If you answered No to the previous question, how likely do you think it is that you might use a programming language in your work in the next year?

very likely	somewhat likely	not very likely	not at all likely

14. How useful do you think programming languages are for your current job, or if you are a student, for your current work at school?

very useful	somewhat useful	not very useful	not at all useful

15. How useful do you think programming languages could be for your work over the next year?

very useful	somewhat useful	not very useful	not at all useful

Stop

Do not go on to the next page.

Appendix C
Summative Assessment Training Instructions

Chart 'n' Art User Study

Introduction to Chart 'n' Art

What is Chart 'n' Art?

You are about to use a prototype drawing program called Chart 'n' Art for the creation of custom charts, graphs, and diagrams which would be difficult to make in standard graphing packages such as Excel, Cricket Graph, or DeltaGraph. I'll ask you to try and do a number of simple charting tasks, but I'm only allowed to give you a small number of preset hints.

Here's an example of a custom chart.

Getting acquainted

First, I'd like you to become familiar with Chart 'n' Art windows and tools. I'll ask you to draw and edit a some simple objects to form a chart.

The basic Chart 'n' Art screen consists of a menubar and a collection of windows.

The Toolbox

The Chart 'n' Art toolbox contains a selection tool and a variety of drawing tools
to create and editing diagrams. You can click on a tool to select it and draw a single shape.

Double clicking locks that tool in place so you can draw many shapes of that kind.

The Color Palette

Choose Green for now.

The Drawing Window

The Chart 'n' Art drawing window is where you create your diagrams. Inside the window are X and Y axes, with the origin at their intersection. (no training condition: ticks mark every 5 units)

Passive & Active conditions: Language Window

The Language Window is where Chart 'n' Art presents information to you about its programming language, SK8Script. This programming language can among other thing help automate the creation of our chart and place chart elements more accurately.
Passive Traning Condition: For now, don't worry about programming language feedback.

Active Training Condition: Pay attention to the programming language information which shows up here and other places on the screen. You'll be using it soon.

Training Task 1: A few simple charts

I'd like you to start by making a simple column chart

1. **Click the Rectangle tool icon in the Toolbox to select it.**
2. **Start drawing a rectangle from 20 units to the right of the origin that is 60 high and 20 wide.**
3. **Start drawing a rectangle from 50 units to the right of the origin that is 20 units high and 20 wide.**
4. **Start drawing a rectangle from 80 units to the right of the origin that is 80 units high and 20 wide.**

You will now create a line chart with three data points

1. **Choose the Line tool in the Toolbox.**
2. **Draw a line from the origin to 50 units above the top of the first column (your first data point.)**
3. **Draw a new line to 80 units above the top of the second column (2nd data point).**
4. **Draw a final line from the end of the second line to 50 units above the top of the third column (3rd data point.)**

I'd like you to spruce up the line chart with some data point symbols.

1. **Choose the Oval tool icon in the Toolbox.**
2. **Position the crosshair just above and to the left of the intersection of the first two lines.**
3. **Draw an oval 5 units high by 5 units wide by dragging diagonally.**
4. **Draw another oval on top of the next intersection.**
5. **Draw one more over the end of the line chart.**

Let's change the colors in your column chart.

1. **Click on the arrow icon in the Toolbox to choose the selection tool.**
2. **Click on the left-most rectangle of the column chart.**

Eight small black squares or "handles" appear on the edges of the rectangle to indicate it is selected. Don't worry if you accidentally move the rectangle.

3. **Click on the swatch of blue in the color palette.**

Select the other two columns and give them the same color.

1. **First, select the middle column.**
2. **Then hold down the shift key and click on the last column.**

Handles appear around both the middle and right-most rectangles, indicating both objects are selected.

3. **Choose blue from the color palette.**

Let's try another color for all three columns.

1. **This time, position the pointer above and to the right of the right-most column.**
2. **Drag diagonally to enclose all three columns with a "marquee."**

All three columns are selected. This is another way to select more than one object.

3. **Change the color to red.**

The values I gave you for the columns were only approximations. Now lets adjust them for their exact values.

1. **Deselect by clicking outside your selection**
2. **Select the left-most column.**
3. **The small black handles mark where you can click to resize it.**
4. **Position the pointer over the middle handle at the top of the column.**
5. **Drag it down to the correct value: 55.**

We also need to change its width to 30.

1. **Drag the middle handle on the right side of the column.**
2. **It should line up directly beside the second column**

Change the width of the middle column so it matches the first column.

The last column needs to be adjusted horizontally and vertically.

1. **Position the pointer over the handle in the top-right corner.**
2. **Drag diagonally down and to the right until it's 75 tall and 30 wide.**

I'd like you to drag the first column to the left so it is flush with the Y axis.

Move it back to where it was before, but this time hold down the shift key.

Notice the shift key constrains your dragging, so the object slides only left and right. The shift key can constrain horizontally or vertically, depending on which direction you first start dragging.

Let's move all three columns to the left simultaneously, so they are next to the y-axis.

1. **Select the columns.**
2. **Drag them to the left.**

Let's add more columns to your chart.

1. **Select the right-most column.**
2. **Choose duplicate from the Edit menu.**
3. **Drag the new column to a into position beside the other columns.**

To save time, we can duplicate more than one shape at a time.

1. **Select the last two columns.**
2. **Choose Duplicate from the Edit menu.**
3. **Drag them into place.**

To make lots of shapes you can use the Duplicate Special menu command.

1. **Start by selecting the last column.**
2. **Choose DuplicateSpecial from the Edit menu.**
3. **Enter 3 into the number of copies box.**
4. **Enter 30 into the horizontal box to place the copies 30 units apart.**
5. **Enter 0 into the vertical box to keep the copies on the Y axis.**

Suppose we want to spread out the columns in the chart.

1. **Drag the right-most column 40 units to the edge of the axis (around the 360 point on the axis).**
2. **Select all of the column by surrounding them with a marquee.**
3. **Choose Distribute Horizontally from the Arrange menu.**

Finally, let's add a background to the chart.

1. Choose the rounded rectangle tool from the toolbox.
2. Draw a rounded rectangle just outside the X and Y axis that encompasses all the elements in your chart.
3. Select it.
4. Choose Send To Back from the Arrange menu to assign it to the back most layer of your drawing.

Passive & Active conditions: Language Window

Training Task 2: Creating bar charts from data sets

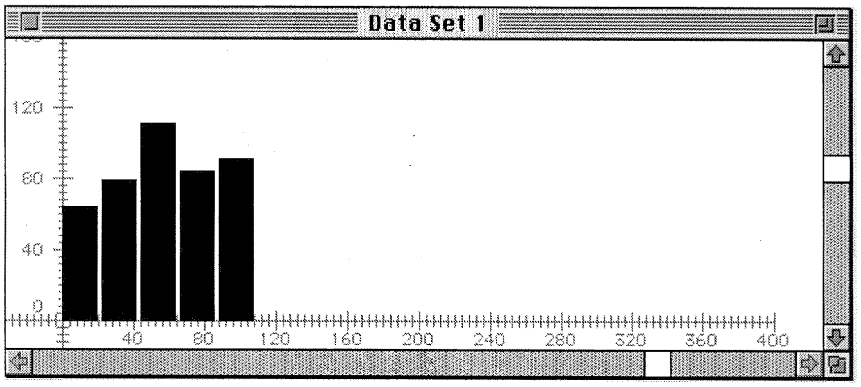
Active Training Condition: Try and use the language as much as possible to make things exact and to help automate repetitive tasks.

Data menu: Data Set 1

Erase your previous drawing using the Erase command from the View menu.

Draw a column chart depicting the information in Data Set 1.

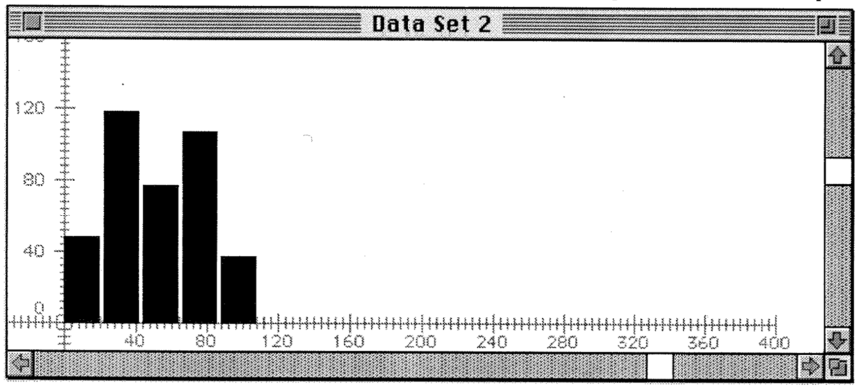
1. Start the first bar at the origin
2. Make the bars 20 units wide
3. Use Duplicate Special to create the remaining rectangles in your chart
4. About bars 20 units apart (usually the same or a little bit bigger than the width)
5. and the length of each bar equal to its corresponding value in the data set.
6. Draw the bars as accurately as possible



Data menu: Data Set 2

Without erasing the drawing, change the length of the bars to reflect the information in Data Set 2.

Active Training Condition: Remember, try and use the language as much as possible to make things exact and to help automate repetitive tasks.

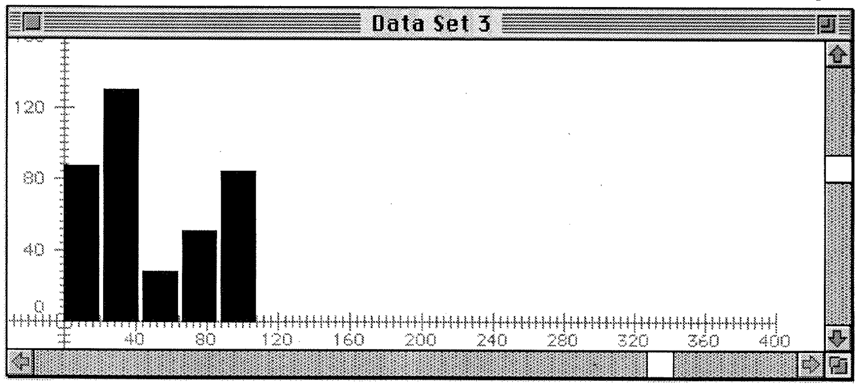


Now make the bars 10 wide instead of 20.

Data menu: Data Set 3

Without erasing the drawing, change the length of the bars to reflect the information in Data Set 3.

Active Training Condition: Remember, try and use the language as much as possible to make things exact and to help automate repetitive tasks.



Now make the bars 15 wide instead of 10.

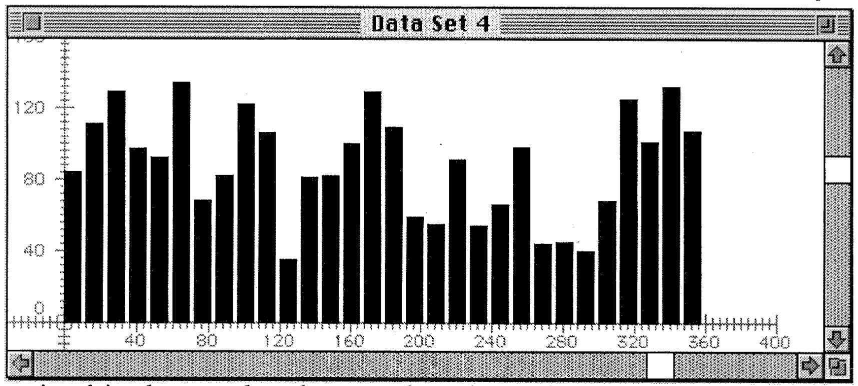
Data menu: Data Set 4

Now erase the drawing and create a new bar chart for Data Set 4.

Start by creating the first bar, then use Duplicate Special to create the remaining copies.

Make the bars about 10 wide and 12 apart.

Active Training Condition: Remember, try and use the language as much as possible to make things exact and to help automate repetitive tasks.



<give hint here only when stuck and user has at least tried 5 bars by hand>

Appendix D
Summative Assessment Introduction to Disclosure Mechanisms and the Language Window

Chart 'n' Art User Study

The Language Window

The Language window is where you often see hints and suggestions for using Chart 'n' Art's programming language. As you have probably already noticed, many operations you perform with the mouse cause CNA to reveal or "disclose" to you how to achieve the same effects with its language.

All conditions:

Start a new drawing by choosing Erase Drawing from the View menu

Create a new oval in the center of the drawing

> **Note the mouse text**
>
> **When you release the mouse CNA elaborates on that information**

The lines in normal type contain actual programming. Text in italics offers an explanation in English of what the expression above it does. More later...

Select the oval

> **Note the object label**

Change its color to Yellow

> **Note the disclosure.**

Make that oval slightly wider by dragging the right-middle handle

> **Note the multiple alternative language expressions**

Create another oval and select it

> **Note the selection disclosure**
>
> **CNA provides a list of expressions which define various properties of the oval.**
>
> **They show values that represent the current state of things on the screen. Note there's a lot of them. Try scrolling down the list**

Select them both

> **Note the object label shows how to refer to both at the same time**
>
> **The selection disclosure now shows expressions for both ovals which define the similarities and differences between properties of these ovals.**

Scroll down and check them out.

The history pane at the bottom allows you to review expressions which have already been disclosed

ordered from top to bottom where the bottom-most entry is the most recent expression shown.

Click on the most recent history entry at the bottom (make Oval)

The expression and any additional information appear now in the white pane.

When write enabled: (active condition in training, all in performance)

Click on the entry for changing the color to Yellow.

The white pane is editable. Change the word Yellow to Green.

Press the Execute Line button

Enter an expression from scratch

Clear

Type in the text at the bottom of the history by refer to the other Oval and use the Pink color.

This time hit the Enter key to execute

If you have multiple lines to execute, you must drag a selection

Can cut and paste within the language window

Appendix E
Hints Given to Subjects during the Summative Assessment

Chart 'n' Art User Study

Hint List

(typically 2 min. threshold, 4 min. when checking manual, or if about to completely abandon strategy)

color > 30	ss chart	train-ing	**Language interface explanations**
			italics indicate an English language explanation of the command above
			object labels show how to refer to one or more objects
			selecting one or more objects discloses a set of "defining" expressions
			history items can be reviewed
			history entries can be reused
			you can edit what's in the white pane of the language window
			to execute a line with the keyboard, hit the enter key
			to execute programming expressions which span multiple lines, first select all the lines
			can cut and paste between data and language windows (after 5 manual text entries)
			option-L generates line continuation character
			arrow keys can be used to scroll horizontally
			Strategy hints
			create the first shape, then use Duplicate Special
			doing something by direct manipulation can often get the system to disclose related language expression
			selecting a single object can provide useful information about what you can do to it with the language
			disclosures from selecting > 1 objects (say by dragging a marquee) can provide useful hints for doing things to multiple shapes at a time (after 5 bars manually)
			to get information on referring to subsets of shapes, try selecting very specific subsets
			it is usually easier to create a shape and then adjust its properties later
			change width with right-middle handle provides simpler way of looking at things
			change height with top-middle handle provides simpler way of looking at things
			don't have to use language

			Language hints
			every item in
			{ } around lists
			bounds = {left, top, right, bottom}
			every item whose height < 20 in ...NOT every item in ... whose height < 20
			‘make’ creates a new shape; it does not update an old one
			height and top are different things
			don’t spell out math terms like ‘<‘, ‘=‘
			Miscellaneous
			to Duplicate Special, enter the # of ADDITIONAL copies
			your drawing doesn’t look quite right according to the my screen snap (when users think they’re done)—might be off by one

Appendix F
Summative Assessment Short-Answer Test

Chart 'n' Art User Study

Written Language Questions

Chart 'n' Art provides a textual programming language for automating the creation of charts and graphs, and helping you place drawing elements more accurately. Almost everything you do with the mouse has equivalent commands in the programming language.

Even if you have never seen Chart 'n' Art's language try and answer the following questions. If you don't know the answer, please ***make your best guess****.*

1. a. Please describe the kinds of *operations* in Chart 'n' Art's language. Operations in a programming language are analogous to verbs in English.

   b. Give examples:

2. a. Please describe the kinds of *operands*, or things which can be operated on, in Chart 'n' Art's language. Operands in a programming language are analogous to nouns and noun phrases in English.

   b. Give examples:

3. a. How do you combine together operations, operands, and other language elements to form Chart 'n' Art commands? Combining operations and operands is like forming a sentence in English.

   b. Give examples:

Stop

Do not go on to the next page.

Appendix G
Summative Assessment Multiple-Choice Test

Note: questions for the actual test were printed one-to-a-page to prevent subjects from refering back to previous questions.

Chart 'n' Art User Study

☐ Multiple Choice Language Questions

Please check your answer for each of the following questions.

1. Chart 'n' Art is designed to teach you about its programming language while you are drawing. How helpful did you find Chart 'n' Art's programming language instruction?

☐ a. `very helpful`

☐ b. `somewhat helpful`

☐ c. `not very helpful`

☐ d. `not at all helpful`

2. How much do you think you learned from Chart 'n' Art's programming language instruction?

☐ a. `a lot`

☐ b. `some`

☐ c. `not very much`

☐ d. `not at all`

*Please check the correct answer for each of the following questions about Chart 'n' Art's programming language. Even if you have never seen the language, please **make your best guess**.*

3. Which expression refers to **the border color of a shape?**

☐ a. `frameColor`

☐ b. `lineColor`

☐ c. `penColor`

4. Which expression refers to **the coordinates of the rectangle bounding a shape?**

☐ a. `bounds`

☐ b. `boundsRect`

☐ c. `frame`

5. Which expression refers to **the thickness of the border of a shape**?

☐ a. `frameSize`

☐ b. `lineSize`

☐ c. `penSize`

6. Which expression refers to **the interior color of a shape?**

☐ a. `fill`

☐ b. `fillColor`

☐ c. `paintColor`

7. Which expression refers to **the thickness of a line?**

☐ a. `lineSize`

☐ b. `lineWidth`

☐ c. `penSize`

8. Which expression refers to **the coordinates of the center of a shape in a drawing window**?

☐ a. `center`

☐ b. `location`

☐ c. `position`

9. Which command **is used to create a shape**?

☐ a. `draw`

☐ b. `make`

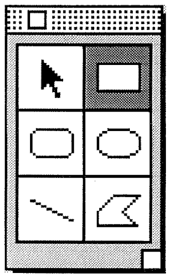
☐ c. `new`

10. Which expression refers to **the window where the graphics appear**?

☐ a. `DrawingWindow`

☐ b. `DrawWindow`

☐ c. `FrontDrawing`

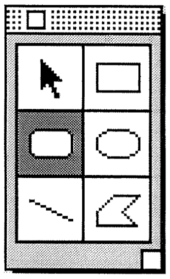


11. **Which expression refers to shapes created by the tool selected in the palette above?**

☐ a. `Box`

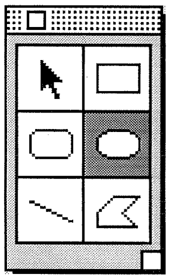
☐ b. `Rect`

☐ c. `Rectangle`



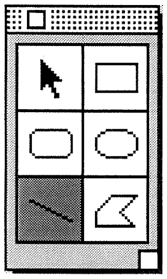
12. Which expression refers to **shapes created by the tool selected in the palette above?**

- ☐ a. `RoundedRect`
- ☐ b. `RoundedRectangle`
- ☐ c. `RoundRect`



13. Which expression refers to **shapes created by the tool selected in the palette above?**

- ☐ a. `Ellipse`
- ☐ b. `Oval`
- ☐ c. `Ovoid`

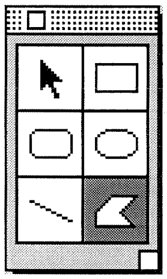


14. Which expression refers to **shapes created by the tool selected in the palette above?**

☐ a. `Line`

☐ b. `LineSegment`

☐ c. `Segment`

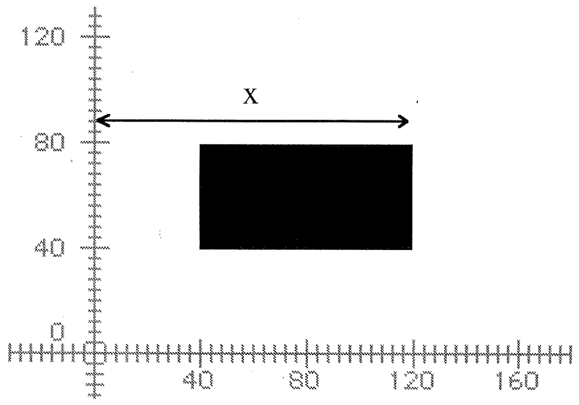


15. Which expression refers to **shapes created by the tool selected in the palette above?**

☐ a. `ClosedPolygon`

☐ b. `Multiline`

☐ c. `Polygon`

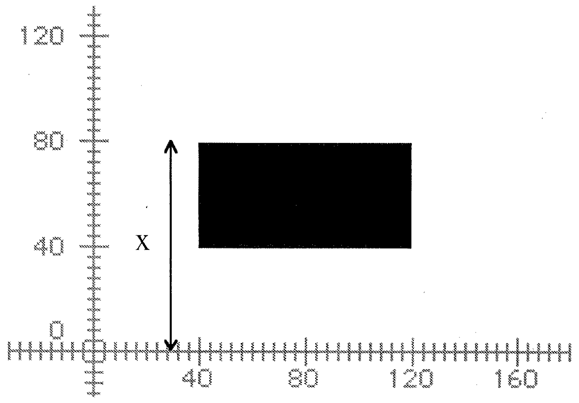


16. Which expression refers to **distance x in the above diagram**?

☐ a. `right`

☐ b. `rightEdge`

☐ c. `xMax`

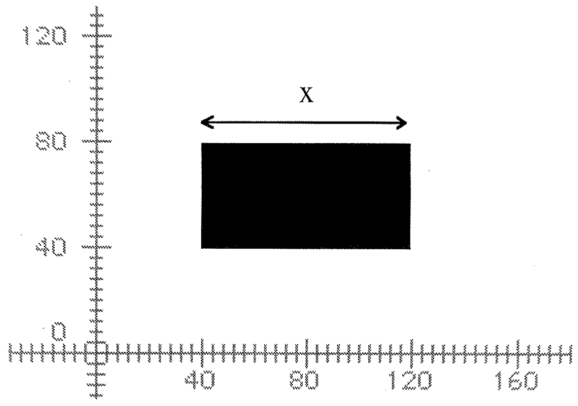


17. Which expression refers to **distance x in the above diagram**?

☐ a. `height`

☐ b. `top`

☐ c. `topEdge`

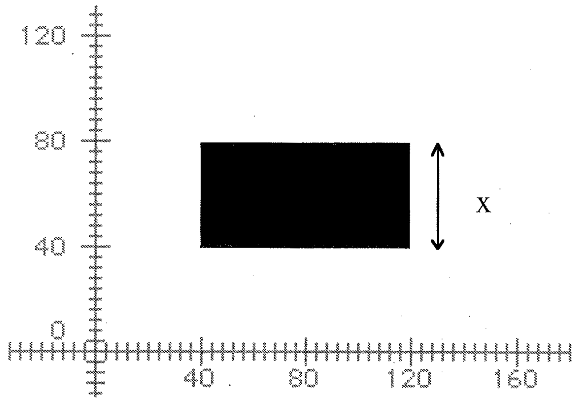


18. Which expression refers to **distance x in the above diagram**?

☐ a. `horizontalSize`

☐ b. `width`

☐ c. `xSize`

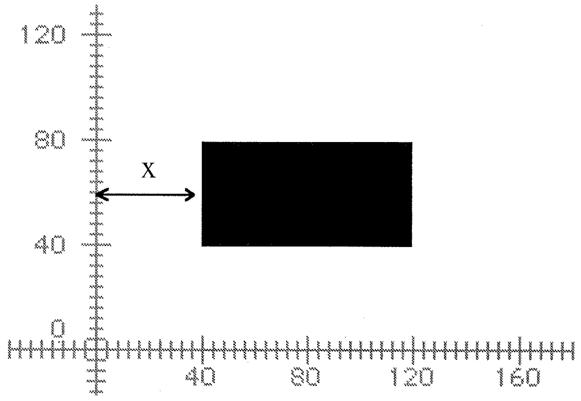


19. Which expression refers to **distance x in the above diagram**?

☐ a. `height`

☐ b. `verticalSize`

☐ c. `ySize`

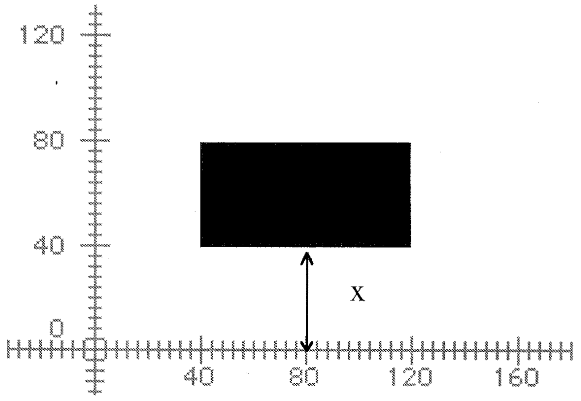


20. Which expression refers to **distance x in the above diagram**?

☐ a. `left`

☐ b. `leftEdge`

☐ c. `xMin`



21. Which expression refers to **distance x in the above diagram**?

☐ a. `bottom`

☐ b. `bottomEdge`

☐ c. `height`

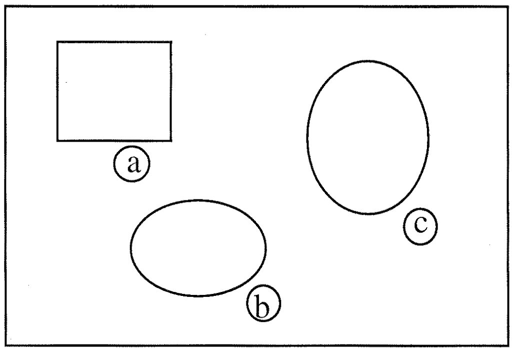
22. Which expression will **draw a rounded rectangle and make its interior color green**?

- ☐ a. `make Green RoundedRect`
- ☐ b. `make RoundedRect fillColor: Green`
- ☐ c. `make RoundedRect whose fillColor = Green`
- ☐ d. `set fillColor Green to make roundRect`

23. Which expression refers to **rectangle number 1 in the drawing window**?

- ☐ a. `find DrawWindow, Rectangle, 1`
- ☐ b. `Rectangle 1`
- ☐ c. `Rectangle 1 in DrawWindow`
- ☐ d. `Rectangle1`

Suppose you drew a rectangle and two ovals shapes into a BLANK drawing window in the order a, b, c as shown below.



24. Which expression refers to **shape** ⓐ ?

- ☐ a. `Rectangle 0 in DrawWindow`
- ☐ b. `Rectangle 1 in DrawWindow`
- ☐ c. `Rectangle 3 in DrawWindow`
- ☐ d. `the Rectangle in DrawWindow`

25. Which expression will **draw a yellow rectangle with a blue border**?

☐ a. `make Rectangle {fillColor: Yellow, frameColor: Blue}`

☐ b.
```
make Rectangle whose fillColor = Yellow and whose ¬
    frameColor = Blue
```

☐ c. `make Rectangle with fillColor Yellow with frameColor Blue`

☐ d. `set frameColor Blue of set fillColor Yellow of make Rectangle`

26. Suppose X refers to a single shape. Which expression will **make that shape 50 units wide by 100 units tall**?

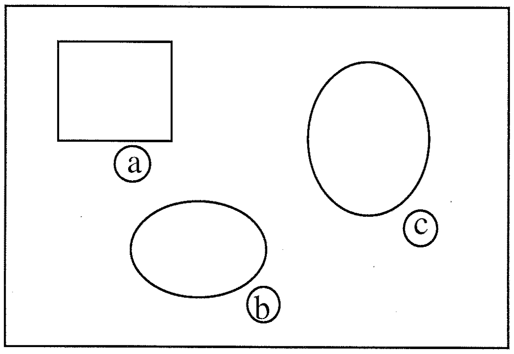
☐ a. `assign the size of X to {50,100}`

☐ b. `set the size of X to {50,100}`

☐ c. `setSize X, 50, 100`

☐ d. `size of X = {50,100}`

Suppose you drew a rectangle and two ovals shapes into a BLANK drawing window in the order a, b, c as shown below.



27. Which expression refers to **the collection consisting of the two objects** (b) **and** (c) ?

☐ a. `every Oval in DrawWindow`

☐ b. `Oval 1 and Oval 2 in DrawWindow`

☐ c. `Oval 1 in DrawWindow and Oval 2 in DrawWindow`

☐ d. `Oval {1,2} in DrawWindow`

28. Suppose X refers to a rectangle centered at {100, 100}. Which expression will **change that shape's location so that its center is now at {0,0}**?

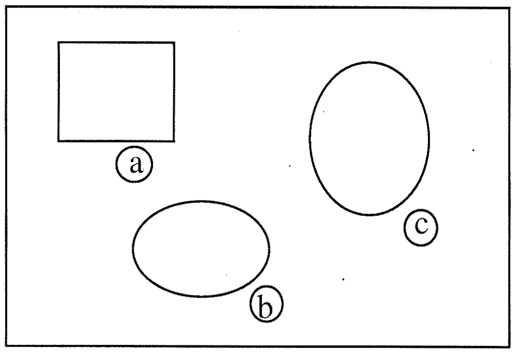
☐ a. `move X by {-100,-100}`

☐ b. `move X to {0,0}`

☐ c. `position X at {0,0}`

☐ d. `setPosition X, 0, 0`

Suppose you drew a rectangle and two ovals shapes into a BLANK drawing window in the order a, b, c as shown below.



29. Which expression refers to **the collection consisting of the two objects** (a) **and** (c) ?

☐ a. `every other item in DrawWindow`

☐ b. `Rectangle 1 and Oval 2 in DrawWindow`

☐ c. `Rectangle 1 in DrawWindow and Oval 2 in DrawWindow`

☐ d. `{Rectangle 1 in DrawWindow, Oval 2 in DrawWindow}`

30. Suppose X refers to a single shape. Which expression will **make that shape's interior color red**?

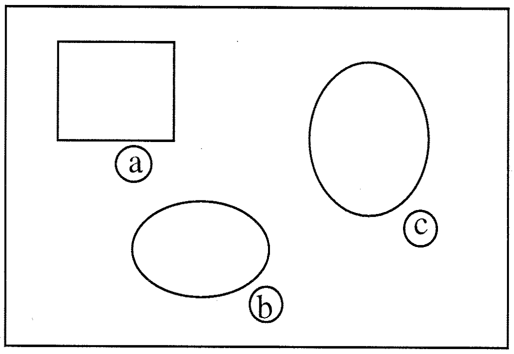
☐ a. `assign the fillColor of X to Red`

☐ b. `fillColor of X = Red`

☐ c. `set the fillColor of X to Red`

☐ d. `setFillColor X, Red`

Suppose you drew a rectangle and two ovals shapes into a BLANK drawing window in the order a, b, c as shown below.



31. Which expression refers to **the collection consisting of the three objects** (a), (b), **and** (c) ?

☐ a. `all shapes in DrawWindow`

☐ b. `all the items in DrawWindow`

☐ c. `DrawWindow contents`

☐ d. `every item in DrawWindow`

32. Suppose X refers to a single shape. Which expression will **make that shape 100 units wide**?

☐ a. `assign the width of X to 100`

☐ b. `set the width of X to 100`

☐ c. `setWidth X, 100`

☐ d. `width of X = 100`

33. Suppose XX refers to a collection of shapes. Which expression will **make the border color of each one of those shapes be pink**?

☐ a.
```
for i = every item in XX
    frameColor of i = Pink
next
```

☐ b. `map the frameColor of XX to Pink`

☐ c. `set the frameColor of every item in XX to Pink`

☐ d. `setFrameColor XX, Pink`

34. Which expression will **hide each rectangle numbered 1 through 10**?

- [ ] a.
```
for i = 1 to 10
   hide Rectangle i in DrawWindow
next
```
- [ ] b. `loop i from 1 to 10 {hide Rectangle i in DrawWindow}`
- [ ] c. `map hide over Rectangle 1 through 10 in DrawWindow`
- [ ] d.
```
repeat with i from 1 to 10
   hide Rectangle i in DrawWindow
end repeat
```

35. Suppose XX refers to a collection of shapes. Which expression will **make the interior color of each one of those shapes be orange**?

- [ ] a.
```
for i = every item in XX
   fillColor of i = Orange
next
```
- [ ] b. `map the fillColor of XX to Orange`
- [ ] c. `set the fillColor of every item in XX to Orange`
- [ ] d. `setFillColor XX, Orange`

36. Suppose XX refers to a collection of shapes. Which expression will **hide each shape in that collection**?

- [ ] a. `dolist XX with i {hide i}`
- [ ] b.
```
for i = every item in XX
   hide i
next
```
- [ ] c. `map hide over XX`
- [ ] d.
```
repeat with i in XX
   hide i
end repeat
```

37. Suppose XX refers to a collection of shapes. Which expression will **make each one of those shapes 100 units tall**?

☐ a.
```
for i = every item in XX
   height of i = 100
next
```

☐ b. `map the height of XX to 100`

☐ c. `set the height of every item in XX to 100`

☐ d. `setHeight XX, 100`

38. Which expression refers to **all of the red shapes in the drawing window**?

☐ a. `all the Red items in DrawWindow`

☐ b. `each item of DrawWindow with fillColor Red`

☐ c. `every item whose fillColor = Red in DrawWindow`

☐ d. `the Red contents of DrawWindow`

39. Suppose XX refers to a collection of five line segments. Which expression will **make the thickness of each of those lines correspond to the values 1, 3, 4, 6 and 7, respectively**?

☐ a. `map the lineSize of XX to everything in {1, 3, 4, 6, 7}`

☐ b.
```
set the lineSize of every item in XX to every item in ¬
   {1, 3, 4, 6, 7}
```

☐ c. `setLineSize XX, {1, 3, 4, 6, 7}`

☐ d.
```
values = {1, 3, 4, 6, 7}
for i = every item in XX
   lineSize of i = values [i]
next
```

40. Suppose XX refers to a collection of shapes. Which expression will **align the left-most edge of each shape with the y-axis**?

☐ a. `alignLeft XX, 0`

☐ b. `align the left of every item in XX to 0`

☐ c. `map the left of XX to 0`

☐ d. `set the left of every item in XX to 0`

41. Suppose XX refers to a collection of five shapes. Which expression will **make the right-most edge of each of those shapes be positioned to correspond to 50, 100, 125, 300, and 360, respectively**?

☐ a. `map the right of XX to everything in {50, 100, 125, 300, 360}`

☐ b.
```
set the right of every item in XX to every item in {50, 100, ¬
   125, 300, 360}
```

☐ c. `setRight XX, {50, 100, 125, 300, 360}`

☐ d.
```
values = {50, 100, 125, 300, 360}
for i = every item in XX
   right of i = values [i]
next
```

The End

Appendix H
Detailed Results from the Summative Assessment

Table H-1. All pair-wise differences between conditions generated using Scheffé post-hoc tests of ANOVAs of short-answer, multiple-choice, and final program scores. For each comparison, the table lists the mean difference, the 95% confidence level, and the p-value. Low experience subjects are those familiar with at most one programming language (E0-1). High experience subjects are those familiar with at least two languages (E2+).

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Short Answer	1.20	0.94	.0106	1.75	0.94	0.0003	0.55	0.94	.3299
Low Exper.	1.96	0.96	.0003	1.86	0.96	0.0005	-0.10	1.04	.9661
High Exper.	0.23	2.15	.9531	1.43	2.15	0.2114	1.20	1.86	.2309
Multiple Choice	12%	16%	.1463	19%	16%	0.0150	7%	16%	.5372
Low Exper.	20%	15%	.0104	15%	15%	0.0489	-5%	17%	.7416
High Exper.	2%	37%	.9845	21%	37%	0.3178	18%	32%	.3002
Final Program	1.05	1.02	.0432	0.45	1.02	0.5262	-0.60	1.02	.3267
Low Exper.	1.36	1.14	.0198	0.66	1.14	0.3215	-0.70	1.24	.3309
High Exper.	0.87	2.22	.5549	0.37	2.22	0.8953	-0.50	1.92	.7634

Table H-2. All pair-wise differences between conditions generated using t-tests of short-answer, multiple-choice, and final program scores. For each comparison, the table lists the mean difference, the 95% confidence level, and the p-value. Low experience subjects are those familiar with at most one programming language (E0-1). High experience subjects are those familiar with at least two languages (E2+).

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Short Answer	1.20	0.93	.0142	1.75	0.81	.0002	0.55	0.65	.0943
Low Exper.	1.96	0.81	.0005	1.86	0.81	.0005	-0.10	0.85	.7924
High Exper.	0.23	2.29	.8116	1.43	2.18	.1575	1.20	0.80	.0085
Multiple Choice	12%	13%	.0625	19%	13%	.0065	7%	13%	.2728
Low Exper.	20%	12%	.0046	15%	14%	.0046	-5%	12%	.3966
High Exper.	2%	34%	.8746	21%	33%	.1732	18%	23%	.1071
Final Program	1.05	0.74	.0079	0.45	0.81	.2557	-0.60	0.94	.1952
Low Exper.	1.36	0.76	.0025	0.66	0.98	.1665	-0.70	1.19	.2110
High Exper.	0.87	1.78	.2790	0.37	1.86	.6458	-0.50	1.64	.5016

Table H-3. T-tests of all pair-wise differences between question topics for each condition. For each comparison, the table lists the mean difference, the degress of freedom, the t-value, p-value, and 95% confidence intervals.

	Mean Diff.	DF	t-Value	P-Value	95% Lower	95% Upper
Vocabulary, Syntax: Total	.095	58	1.972	.0534	-.001	.191
Vocabulary, Syntax: No Training	.226	18	2.956	.0085	.065	.387
Vocabulary, Syntax: Passive Training	.047	18	.680	.5052	-.099	.194
Vocabulary, Syntax: Active Training	.011	18	.137	.8925	-.151	.172
Vocabulary, Semantics: Total	.140	58	3.198	.0022	.052	.228
Vocabulary, Semantics: No Training	.173	18	2.381	.0285	.020	.325
Vocabulary, Semantics: Passive Training	.152	18	2.360	.0298	.017	.287
Vocabulary, Semantics: Active Training	.097	18	1.187	.2506	-.075	.268
Syntax, Semantics: Total	.046	58	.857	.3950	-.061	.152
Syntax, Semantics: No Training	-.054	18	-.677	.5072	-.220	.113
Syntax, Semantics: Passive Training	.104	18	1.301	.2097	-.064	.273
Syntax, Semantics: Active Training	.086	18	.979	.3406	-.099	.272

Table H-4. All pair-wise differences between conditions generated using Scheffé post-hoc tests of ANOVAs of scores for three types of questions on the multiple-choic test. For each comparison, the table lists the mean difference, the 95% confidence level, and the p-value.

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Vocabulary	0.105	0.165	.2721	0.084	0.165	.4286	-0.021	0.165	.9469
Syntax.	0.142	0.217	.2550	0.300	0.217	.0053	0.158	0.217	.1886
Semantics	0.160	0.211	.1649	0.160	0.211	.1649	0	•	•

Table H-5. All pair-wise differences between conditions generated using t-tests of scores for three types of questions on the multiple-choic test. For each comparison, the table lists the mean difference, the 95% confidence level, and the p-value.

	Active Training - Passive Training			Active Training - No Training			Passive Training - No Training		
	Mean Diff.	+/-	p	Mean Diff.	+/-	p	Mean Diff.	+/-	p
Vocabulary	0.105	0.13	.0993	0.084	0.15	.2400	-0.021	0.13	.7331
Syntax.	0.142	0.18	.1085	0.300	0.18	.0021	0.158	0.18	.0758
Semantics	0.160	0.18	.0739	0.160	0.18	.0739	0	•	•

Table H-6. Means for the multiple-choice test scores grouped by question topic and condition. The "Raw" scores are those which include all valid questions on the test. The "Adjusted" scores are those which exclude 11 questions potentially influenced by the disclosures exercises. For each mean the table lists the 95% confidence, standard error, and standard deviation.

	Raw			Adjusted		
	No Training	Passive Training	Active Training	No Training	Passive Training	Active Training
Vocabulary Mean	45%	43%	54%	37%	35%	47%
Conf. Level	11%	8%	11%	12%	7%	12%
Std. Err.	0.049	0.036	0.049	0.052	0.029	0.054
Std. Dev.	0.155	0.113	0.155	0.164	0.093	0.169
Syntax Mean	23%	38%	53%	20%	30%	41%
Conf. Level	13%	14%	13%	14%	10%	14%
Std. Err.	0.059	0.060	0.059	0.060	0.043	0.063
Std. Dev.	0.186	0.189	0.187	0.189	0.137	0.198
Semantics Mean	28%	28%	44%	13%	13%	30%
Conf. Level	12%	12%	15%	13%	13%	18%
Std. Err.	0.053	0.053	0.065	0.056	0.056	0.082
Std. Dev.	0.169	0.169	0.207	0.177	0.177	0.258
Overall Mean	34%	41%	53%	30%	33%	44%
Conf. Level	10%	10%	10%	10%	7%	11%
Std. Err.	0.043	0.043	0.045	0.044	0.033	0.048
Std. Dev.	0.136	0.135	0.143	0.140	0.103	0.151