Collaborative Applications for Long-Term Use

Michael Eisenberg
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder
Boulder, CO 80309-0430
Email: duck@cs.colorado.edu

Abstract

Computer applications are designed, used, and assessed by (often unstated) analogy to other tools and devices. When viewed as appliances (e.g., telephones), applications are created and judged according to standards of ease of use and rapid learnability; when viewed, by contrast, as creative media (e.g., musical instruments), standards of expressiveness and flexibility become more salient. In designing this second type of application, providing the user with long-term creative power often necessitates an acceptance of various forms of complexity within the application—for example, the application may include a "task-enriched" programming environment; and further, making such an application truly collaborative suggests the inclusion of features and tools that will support users in learning both the application itself (including the programming language) and the domain of the application. This paper elaborates on these observations, discussing the design of collaborative applications for creative, long-term use (and contrasting this design strategy with the dominant current trend in application development). The paper concludes with a brief description of our own efforts in collaborative application design: namely, the *programmable design environments* currently being prototyped in our laboratory.

Introduction

Personal computer software, as a phenomenon, is now—depending on one's measure of these things—about twenty years old. But for its users, designers, and critics—for those burgeoning numbers of people whose day-to-day lives center upon computers—there is a continuing struggle to define what, precisely, computer software is *for*. That struggle often takes the form of conflicting metaphors, answering the question of what software is for by focusing on what it is like. Thus, on one view, software is like the written word, destined to be the carrier of a popular intellectual revolution similar to that sparked by the printing press [Friedlander 1988, Kay 1993]; or it's like a physical environment, a virtual world, in which users can move and interact [Rheingold 1991]; or it's like movies or theater or television—entertainment of a sort—but ideally less passive than those other media [Crockford 1990, Oren 1990, Laurel 1991]; or it is a tool, to be integrated seamlessly and "conversationally" into day-to-day activity [Winograd and Flores, 1986]; or, perhaps more ominously, it is like the written word *before* the printing press, an invention that separates the world into a population of elite scribes and a much larger population of intellectually disenfranchised computerphobes. [Fischer 1993]

All these metaphors have compelling arguments to accompany them; and all provide fascinating insights by focusing our attention on important aspects of computer software. But these varying metaphors—appropriate as they may be for different circumstances—do lead to distinct and sometimes conflicting philosophies of software design and use. For application software in particular, this welter of interpretive metaphors can prove confusing: do we wish to design (and judge) applications as entertainment, as handy task-specific tools, as means of expression, or as something else entirely? This basic task—of understanding what applications *are* , how we want to think about them—takes on a new urgency as the sheer quantity, variety, and complexity of software applications becomes overwhelming.

This paper will use this notion of "interpretive metaphors" to focus on several issues that I believe to be recurrent in the design of expressive software applications. A central theme, cropping up in one guise or another throughout the discussion, is the question of pace, the time frame over which software is used. If software is designed to be encountered briefly, intermittently, to perform relatively predictable or constrained jobs (or jobs for which failure would be catastrophic), then design criteria of simplicity and learnability become paramount. (The operant metaphor here might be one which likens computer software to near-ubiquitous appliances like the telephone, refrigerator, or automobile.) On the other hand, if software is designed with the notion of long-term individual human-computer collaboration in mind, then complexity becomes harder to avoid (perhaps it is no longer worthwhile to avoid it altogether), and expressiveness is the most important dimension of design. (The operant metaphor here might be one which compares computer software to the medium of writing, or to musical instruments.)

The first section of this paper pursues the theme of the previous paragraph by focusing on the tension between learnability and expressiveness in the design of collaborative applications. This is followed by a discussion, in Section 2, of how the current culture of software development might be viewed as antagonistic toward achieving the goal of human-computer

collaboration. The third (and final) section of the paper briefly describes some current work in our laboratory at the University of Colorado, and the issues that this work raises for prospects of developing truly collaborative software applications.

Learnability vs. Expressiveness

In keeping with the tone of the introductory paragraphs above, it may be worth beginning this section with an analogy. Imagine that we are presented with two brand-new artifacts, never before seen: the first is a keyboard-operated musical instrument called a "piano,'" and the second is a similar but slightly enhanced tool, the "player piano." A user of the first instrument is expected to spend months, or perhaps years, mastering the interface; although the user may, just barely, be able to coax listenable music out of the piano during this learning period, it will almost certainly take years before he or she can expect to play (or compose) anything of truly high quality. In the second case, however, the task is much easier: the user simply buys a set of available piano rolls, plugs them into the slot at the front of the instrument, and listens to the competent (if perhaps less than brilliant) music produced.

Certainly both these instruments have their strengths; but those whose values are derived from the human-computer interaction community might be inclined to judge the piano rather harshly. The interface is, after all, difficult to master—the learning curve could hardly be steeper. Indeed, out of a hundred beginning piano students, perhaps only a small number will persevere long enough to create listenable music; many more will give up and move on to other pursuits. If we think of the piano, then, as a tool with which to produce music, it must be admitted that it is a rather inefficient one: slow, uncertain, and demanding. In contrast, the player-piano seems to be a distinct improvement: if the user has to produce music right away, the appropriate piano roll is obtained. The interface to the instrument virtually guides the user into the music-making process.

The contrast is, of course, incomplete as stated: while the piano is admittedly a difficult and frustrating instrument, it does offer—to those who finally succeed in mastering it—a wonderfully expressive medium. Even for those of us who do not master the instrument (or even attempt it), the piano provides us with enjoyment through the creative efforts of those who do. In contrast, the player piano offers users a quick-and-easy selection. While enjoyable and worthwhile as a tool for the rapid recreation of music, it is hardly a means of expression; and while learnable, it simultaneously diminishes the role and rewards of expertise.

The contrast between the piano and player piano has some analogy—an imperfect analogy, granted, but still illustrative—with issues that arise in applications software. Loosely, we might equate the player piano example with efforts to create "user-friendly" software whose interface is entirely driven by menus, dialog boxes, icon selection, and so forth. Certainly the best software of this type is learnable, and—like the player piano—it guides the user into a relatively painless selection among designer-provided alternatives. On the other hand, there is little reward for expertise in such systems: while the experienced user may become familiar with a wider array of options (analogous to the player piano user acquiring a larger set of piano rolls from which to choose), the development is one of "breadth" rather than "depth." That is, one knows more and more about the myriad features of the software, but the internal computational world—the way in which the computer can serve to reinvent the application's domain—remains mysterious, the province of the designer.

In contrast, programming languages are closer in spirit to the piano. For many people, they require a long time to learn; the initial rewards of that learning process are small; but the range of expressiveness that accompanies expertise is tremendous. Perhaps most telling of all, programming (like the piano) can become a lifelong passion—an activity that takes on more value over the course of a lifetime precisely because we have been growing with it for years. Though the essential rules of programming may be learned in a brief time (much like the rules of chess, or the placement of keys on the piano), it is the progressive, long-term combination of those rules in creative, personalized patterns that gives the programmer his or her voice.

In point of fact, one can argue that the design values represented by the piano and player piano—namely, expressiveness and learnability, respectively—need not be cast in opposition in the realm of application software. Ideally, software can be both learnable and expressive.[Eisenberg 1991] While this line of argument could be pursued at length, the point worth making here is that expressiveness is the more fundamental value for long-term applications: the question of whether software is learnable should be viewed as ancillary to the deeper question of what one can do, eventually, with the software. And again, the issue of timing is crucial: expressive software can and should be designed to foster life-long passions, life-long development, and a sense of contemplation (or what Norman [1993] refers to as "reflective cognition"), even if this should come at some cost in immediate gratification.

The Frantic Culture of Software Development

The brief discussion of the previous section admittedly caricatures many of the design questions surrounding application software. But the issue of time frame does come up—sometimes vividly—when looking at descriptions of current application software. At least a half dozen monthly magazines are now in large part devoted to software reviews, and some of these reviews are telling. One recent magazine review of a paint program noted that the latest release of the program had added 70 new features.

Other reviews from the past several months' crop of magazines include the following statements:

> "XyWrite 4.0... offers improvements in performance and dozens of new commands and ease-of-use features." [PC World May 93 p.82]

> "Publish It! Easy is an entry-level desktop-publishing program packed with page-layout features you'd expect to find only in packages costing three times the price... [It] offers a wider range of paint and drawing tools than in previous version..." [Mac Computing special issue 1993]

> "The latest version of Theorist features several important enhancements... In addition to adding support for tables, the new version of Theorist provides QuickTime support... Other new features include support for Fourier transforms and Bessel functions, the ability to create scatterplots from tables and matrices, and the ability to customize the appearance of notebooks with different fonts." [MacUser June 1993]

> "MacProject Pro is a significant upgrade to MacProject II, providing tools for better organization and entry of data, enhanced display capabilities, much-improved resource management, more-sophisticated scheduling features..." [MacWorld May 1993]

Depending on one's view—maybe depending on one's mood—there is good news or bad in these reviews. On the one hand, they reflect an astonishing pace of development in application software, and a similarly astonishing outpouring of creativity in design. And perhaps they can also be viewed as representative of an ever-growing range of possible activities for computer users. But there is a disturbing note as well, a frantic and unreflective note: will the ninth iteration of our paint program eventually include 500 more features, and if so how will we ever accommodate them into our own work? And will those 500 new features really improve the software, and really improve our understanding of graphics or design or whatever—or will they simply overwhelm us with choices that we can never explore? Might it be that the accretion of features, motivated in part by the need to sell ever-newer versions of software, proves counterproductive for the long-term user?

It may be worth returning once more, briefly, to the piano analogy. Would we really be better off if, every ten months, a new version of the piano was introduced—a new version largely incompatible with the previous one, and many more options added? Could such an environment foster another Beethoven? Or—to put the question another way—would the pace of this production really be compatible with the pace of human creative development?

The current culture of software development places a heavy emphasis on the elaboration of large, varied, and extensive feature sets. Programs are compared to their competitors (and to their previous incarnations) according to how many new features have been included; presumably, the more the better. Clearly some of these features will prove useful; but collectively, they signal a troubling trend in the development of collaborative applications. On the one hand, they promote a style of use in which the endless exploration of new features takes on more importance than the patient creative mastery of an expressive medium lending itself to abstraction and composition; at the same time, they distract designers' attention from the more fundamental task of construcing tools that offer the possibility of long-term growth, education, and creativity. The impression ostensibly conveyed by the popular computer press is one of a plethora of ever-expanding choice; yet the feeling that results is one not of possibility but of overload, similar to the frantic portrait of multichannel cable television portrayed by [McKibben 1993]:

> "Type A personalities are five times as likely to have a *second* heart attack, according to Otto Wahl, the psychiatry professor at George Mason University. Following vertical roasting on the Spanik Vertical Roaster, a chicken can be—is—carved with a carrot. In Czechoslovakia, Ambassador Rita Klimova tells C-SPAN, the newly emerging democracy has spawned dozens of political parties, including one for beer drinkers. Sesame Street is brought to you this morning by L, S, and 6. Only 11 percent of Americans feel the penny should be banned..."

Programmable Design Environments

In our laboratory at the University of Colorado, we are currently developing prototype applications whose purpose is to address (among others) the issues raised in the previous discussion. These *programmable design environments* are intended to integrate aspects of programmability, direct manipulation interfaces, critiquing and design rationale capabilities, and supportive learning environments.[Fischer and Eisenberg 1992] The first of these applications is a graphical design system primarily aimed at the creation of charts and information displays; the second is an application for the design and simulation of local-area computer networks.

These two applications are designed as models of collaborative software in which the computer system takes on certain specific (and, we believe, appropriate) roles: presenting a learnable (and aesthetically attractive) interface, a full-fledged programming environment, a catalog of previous work done with the application, and critiquing/tutorial components that act both to help users in coping with the complexity of the domain (graphic or network design) and with the complexity of the application. Because these applications are designed to include domain-specific programming environments as well, they likewise address the issue of expressiveness mentioned earlier. (Cf. the arguments in [Nardi 1993].) By

presenting the user with a programming environment enriched with appropriate primitive procedures and objects as building blocks, these systems allow the user to develop a personalized vocabulary that can grow in complexity over time—conceivably (and ideally), over a lifetime. (The portrait of the programming process suggested by these applications is consistent with the picture of "metalinguistic abstraction"— of using languages to build still other domain-enriched languages—eloquently articulated by Abelson and Sussman [1985].)

Our graphic design application—whose current prototype is built on top of the SchemePaint system [Eisenberg 1991]—includes a (still rudimentary) direct manipulation interface for selecting the type of chart that the user wishes to create (e.g., bar chart, line chart, scatter plot, and so forth); having selected this type of chart, the user is able to look at a variety of relevant examples built using the system, and employing language primitives (embedded in the Scheme programming language) appropriate to building this particular kind of chart. For instance, a user who wishes to create a bar chart is presented with a palette showing a variety of "specialty" bar charts (multicolored bars, bars with non-horizontal uppermost lines, bars going above and below the horizontal axis, etc.); the user can then access a language tutorial specifically geared to this particular type of chart (e.g., what language form was used to create this particular sample), and can access additional tutorials on the proper use and critiquing for this type of chart (that is, when this sort of graph might be useful, or counterproductive, and why). In presenting the user with a variety of examples (chosen for their pedagogical or illustrative value), the application is supplying some context-driven "feature" presentation; that is, the system is providing the user with some (presumably popular) alternatives for creating new charts. On the other hand, because the system presents not merely a choice of existing styles of work, but rather an extensive programming environment, it gives the user the (more difficult but ultimately more powerful) option of developing new types of charts on his or her own; moreover, it provides the user with assistance in developing expertise in the system-supplied language.

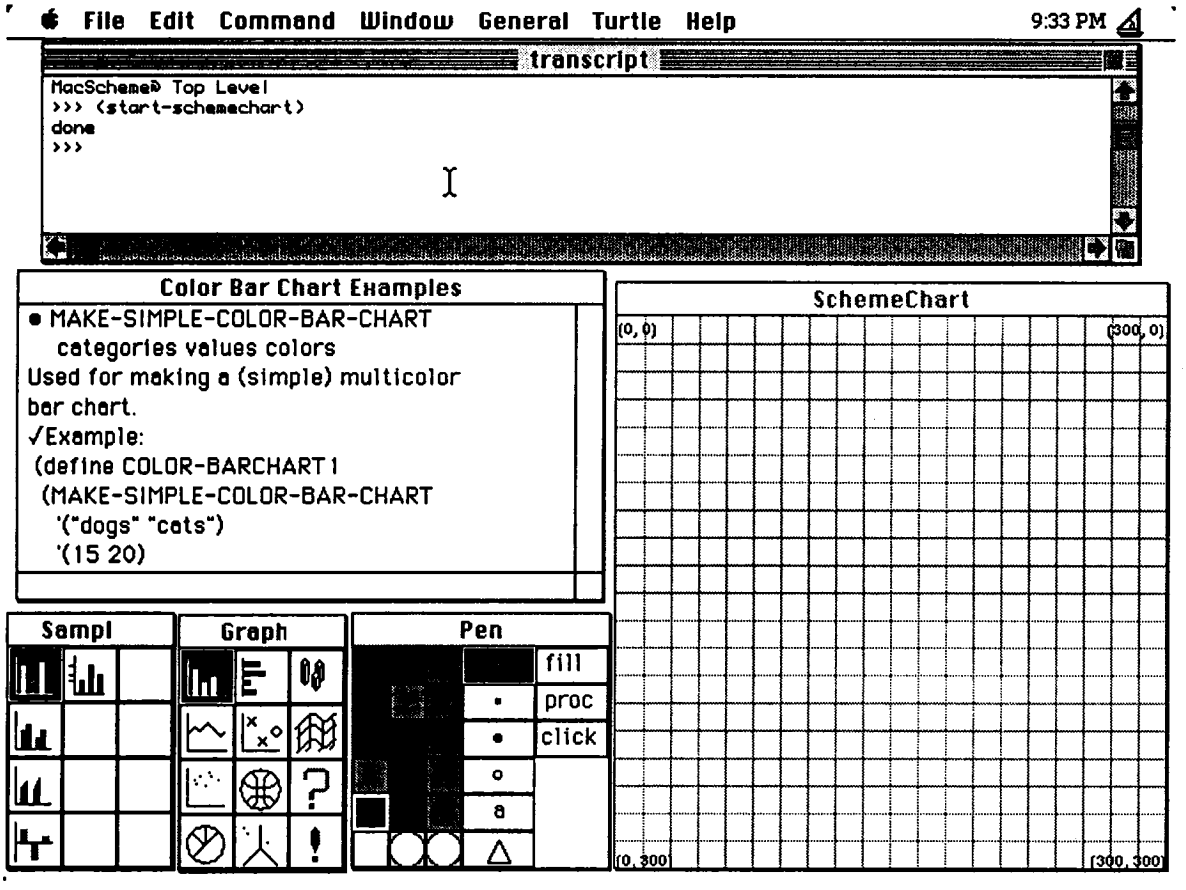

Figure 1. A screen view of the graphic design application. Charts are created in the **SchemeChart** window at bottom right; the **Pen** window provides standard paint tools; the **Graphs** window provides an overview of standard chart types from which to choose; and, for a given graph choice, a catalog of specific examples is provided in the **Samples** window. Finally, the **transcript** window at top provides an "application-enriched" Scheme interpreter. [The system is written in MacScheme (Lightship Software, Mountain View, CA) and runs on all Apple Macintosh computers.] In this figure, the user has selected a particular type of bar chart and has used a menu option to access relevant programming examples for this type of chart; the examples are shown in the window labelled **Color Bar Chart Examples.**

Figure 1 depicts a screen view of the application in the course of a typical chart-design task: here, the user has employed the iconic catalog of bar chart designs (in the **Samples** window at bottom left) to select a particular type of bar chart. She then uses a menu selection to bring up a window in which a number of relevant language forms—procedures that can be used to generate this type of chart—are illustrated. By editing and evaluating these examples directly, the user can create her own charts as modifications of the existing types; after acquiring greater expertise with the language, the user can ultimately access the definitions of these language forms and create new types of charts within the application.

Figure 2 shows another technique for exploring the language environment within the application; here the user has selected "query mode" (the question mark icon in the **Graphs** window), which allows the user to select graph elements from an already-created graph and to access procedures relevant to the creation or change of those graph elements. In the figure, the user has "queried" the vertical axis of a newly-created graph; and the system responds with a (still rather skeletal) list of procedures that are especially useful in altering chart axes. (This feature is very similar to the example-driven software-reuse system of Redmiles [1993].)

The second application—for network design—is built in Repenning's Agentsheets system [Repenning 1993] and HyperCard, and will likewise include a programming environment for simulation of networks; the current prototype focuses on providing a rapid, computer-assisted construction environment for network models (the Agentsheets portion of the system), and on providing tutorials in the theory of network design (the HyperCard portion of the system). Indeed, in both these applications there is an emphasis on the notion of *learning on demand* [Fischer 1991] within the system; the applications are created to encourage a seamless transition between design work (e.g., the user creating a network or chart) and progressive learning about the system and the domain (through catalogs, critics, and embedded tutorials).

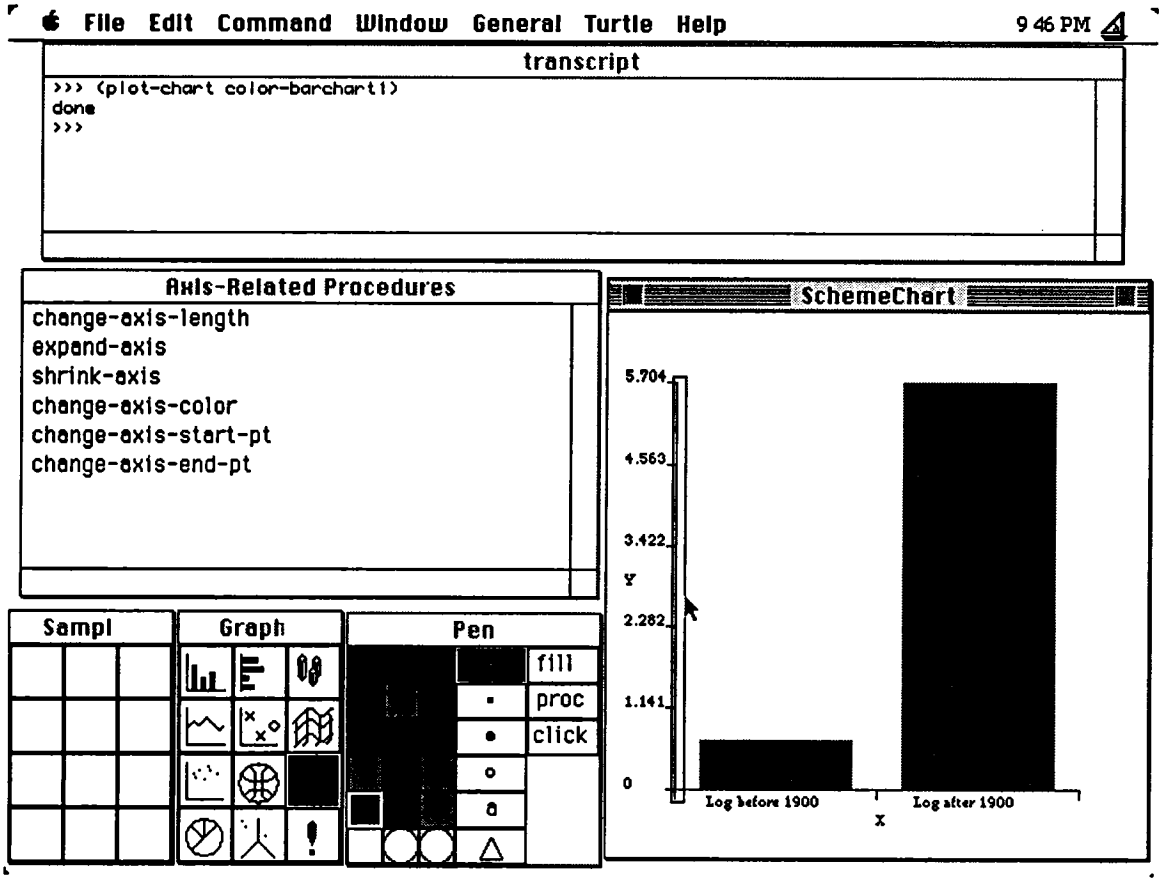

Figure 2. Here, the user has selected "query mode" (the question mark icon) from the **Graphs** window. By dragging the mouse over a particular element of the newly-created graph (in this case, the y-axis), the user can access a list of language procedures relevant to the manipulation of this element.

Acknowledgments

The work described in this paper was supported by NSF grant MDR-9253425 and by NSF Young Investigator Award IRI-9258684. The author also gratefully acknowledges the influence of Harold Abelson, Gerhard Fischer, Clayton Lewis, and Gerald Sussman, as well as that of the Human-Computer Communications group at the University of Colorado.

References

Abelson, H. and Sussman, G. with Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA 1985.

Crockford, D. "Integrating Computers and Television." In *The Art of Human-Computer Interface Design*, B. Laurel, ed. Addison-Wesley, Reading, MA 1990.

Eisenberg, M. Programmable Applications: Interpreter Meets Interface. MIT Artificial Intelligence Laboratory Memo 1325, October 1991.

Fischer, G. "Supporting Learning on Demand with Design Environments." In *Proceedings of the International Conference on the Learning Sciences*, Evanston, IL 1991.

Fischer, G. "Beyond Human-Computer Interaction." To be published in *Mensch-Computer Interaktion auf dem Weg in die industrielle Praxis*, H-D. Boecker and T. Strothotte, eds. Springer-Verlag, Heidelberg, 1993.

Fischer, G. and Eisenberg, M. "Programmable Design Environments." University of Colorado Computer Science Technical Report CU-CS-620-92. October 1992.

Friedlander, L. "The Shakespeare Project." In *Interactive Multimedia*, S. Ambron and K. Hooper, eds. Microsoft, Redmond WA 1988.

HyperCard. (software) Apple Computer, Inc. Cupertino, CA.

Kay, A. In *Watch What I Do*, A. Cypher, ed. MIT Press, Cambridge MA. 1993

Laurel, B. *Computers as Theatre.* Addison-Wesley, Reading, MA. 1991.

McKibben, B. *The Age of Missing Information.* Penguin Books, New York 1993.

Nardi, B. *A Small Matter of Programming.* MIT Press, Cambridge, MA 1993.

Norman, D. *Things That Make Us Smart.* Addison-Wesley, Reading, MA 1993.

Oren, T. "Designing a New Medium." In *The Art of Human-Computer Interface Design*, B. Laurel,ed. Addison-Wesley, Reading, MA 1990.

Redmiles, David. "Reducing the Variability of Programmers' Performance Through Explained Examples." *Proceedings of INTERCHI '93*, Amsterdam.

Repenning, A. "Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments." *Proceedings of INTERCHI '93*, Amsterdam.

Rheingold, H. *Virtual Reality.* Touchstone, New York 1991.

Winograd, T. and Flores, F. *Understanding Computers and Cognition: a New Foundation for Design.* Ablex, Norwood, NJ. 1986.