Programmable Design Environments

Michael Eisenberg and Gerhard Fischer

CU-CS-620-92 October 1992

University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-7514, FAX: (303) 492-2844

Programmable Design Environments

Michael Eisenberg and Gerhard Fischer
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder
(duck, gerhard)@cs.colorado.edu; (303) 492-8091

Submitted to INTERCHI'93

Abstract

Programmable design environments (PDEs) are computational environments that integrate conceptual frameworks and system components developed independently in research efforts on (a) programmable applications and (b) design environments. The integration of these two approaches eliminates their individual weaknesses by simultaneously supporting expressiveness, assistance, modifiability, and domain-oriented descriptions. A sequence of scenarios from the domain of graphic arts is used to illustrate a pattern of "design--assessment/evaluation--redesign" in the construction of applications; in following this sequence, we progress from separate systems that support programmability, direct manipulation, construction and argumentation to an integrated system architecture combining the strengths of all of these approaches.

The research was supported by the National Science Foundation under grants IRI-9015441 and MDR-9253425. The first author is supported in addition by a National Science Foundation Yourg Investigator grant (IRI-9258684).

Acknowledgements. The authors would like to thank the members of the Human-Computer Communications group at the University of Colorado who contributed to the conceptual frameworks and systems discussed in this article.

Programmable Design Environments

Michael Eisenberg and Gerhard Fischer

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder CO 80309
(duck, gerhard)@cs.colorado.edu; (303-) 492-8091

ABSTRACT

Programmable design environments (PDEs) are computational environments that integrate conceptual frameworks and system components developed independently in research efforts on (a) programmable applications and (b) design environments. The integration of these two approaches eliminates their individual weaknesses by simultaneously supporting expressiveness, assistance, modifiability, and domain-oriented descriptions. A sequence of scenarios from the domain of graphic arts is used to illustrate a pattern of "design—assessment/evaluation—redesign" in the construction of applications; in following this sequence, we progress from separate systems that support programmability, direct manipulation, construction and argumentation to an integrated system architecture combining the strengths of all of these approaches.

KEYWORDS

Direct manipulation, end-user programming, programmable applications, domain-oriented construction kits, argumentative hypermedia, design rationale, critics, integrated design environments, programmable design environments, graphic arts.

INTRODUCTION

Software applications have in recent years become crucial and ubiquitous tools for professionals in a variety of complex domains. Architects, electrical engineers, chemists, statisticians, cognitive psychologists, and film directors (among many others) all now depend for their livelihood on the mastery of various collections of applications. These applications, in order to be at all useful, must provide domain workers with rich, powerful functionality; but, in doing so, these systems likewise increase the cognitive cost of mastering the new capabilities and resources that they offer [11]. Moreover, the users of most of these applications soon discover that "software is not soft": i.e., that the behavior of a given application cannot be changed or meaningfully extended without substantial reprogramming. The result is that most applications offer only a rather illusory and selective power: new users are not provided with support in learning and mastering the features of the application, while experienced users are not given the expressive range needed to augment, personalize, and rethink those features.

Over the last few years, we and other researchers have developed conceptual frameworks and innovative systems to address this problem. In this paper, we describe the evolution (driven by the assessment and critical evaluation of our previous efforts) towards programmable design environments integrating two software design paradigms that we have each propounded separately—namely, integrated domain-oriented design environments [8, 9, 10] and programmable applications [7]. We will illustrate the evolutionary development of programmable design environments using an example from the domain of graphic arts [24].

THE ROAD TO PDES (1): CONCEPTUAL FRAMEWORKS AND SYSTEM-BUILDING EFFORTS

Figure 1 on the following page provides an overview of the historical path towards programmable design environments (PDEs). The two basic constituents of PDEs, programmable applications and domain-oriented integrated design environments, were each themselves the result of integrating two separate paradigms. The paradigms serving as ancestors will be briefly discussed, and their strengths and weaknesses will be illustrated (especially in the context of the scenarios of the following section).

Programmable Applications

Programmable applications are systems that combine direct manipulation interfaces with interactive programming environments. The direct manipulation portion [23,17] of the application is designed to help users

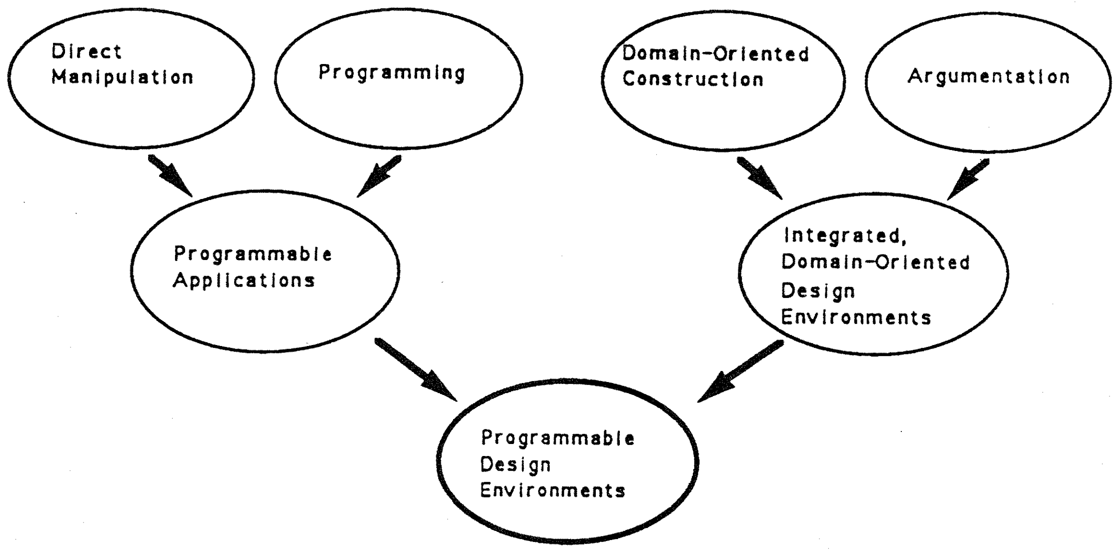

Direct Manipulation
Programming
Domain-Oriented Construction
Argumentation
Programmable Applications
Integrated, Domain-Oriented Design Environments
Programmable Design Environments

Figure 1: The development history of programmable design environments

explore the basic functionality of the system and, perhaps more important, to give users the opportunity to employ their "extra-linguistic" skills of hand-eye coordination. The programming environment [1, 4, 6, 21] is designed to provide users with extensibility and expressive range. This portion of the application is constructed around a domain-enriched language (which might be a newly-constructed language, or an application-specific "dialect" of some existing general-purpose language); the essential design principle behind this language is that users should be able to express interesting ideas within the application domain merely by writing short, simple programs [14].

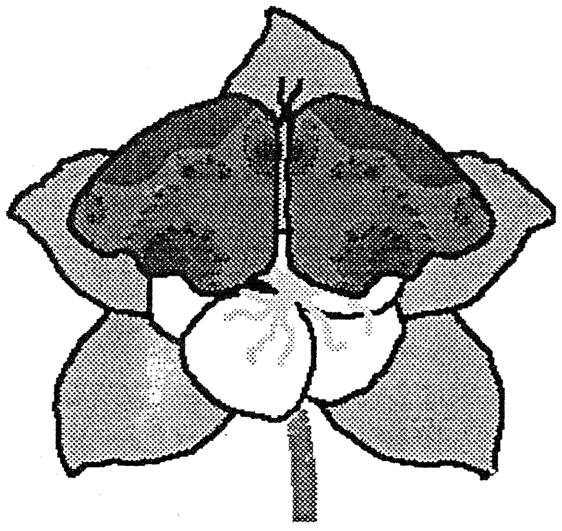


Figure 2: A SchemePaint picture.

SchemePaint [7] is both a working prototype and illustration of a programmable application: it is a graphics application that combines a "Macintosh-style" direct manipulation interface with a graphics-enriched Scheme programming environment [6]. The artist employing SchemePaint can create graphics effects both "by hand" (using the mouse in drawing mode) and "by code" (that is, by writing graphical Scheme programs), and can combine these elements within single drawings. The use of SchemePaint has shown that artists can create works that would be near-impossible to achieve either by "pure" direct manipulation or by "pure" programming alone. Figure 2 shows an example: here, a butterfly with fractal wings (drawn by a program) is posed upon a largely hand-drawn flower.

Weaknesses of Programmable Applications

While programmable applications overcome the limitations of stand-alone direct manipulation systems and end-user programming environments (for a more detailed analysis see [7]) by providing ease of use and expressiveness simultaneously, they have their own characteristic shortcomings. Three of the most prominent of these are: (1) programmable applications are still not sufficiently domain-oriented (i.e., the conceptual gap between the computational substrate provided and artifacts constructed is still too large), (2) they provide insufficient support and feedback to achieve quality artifacts (i.e., the use of SchemePaint has shown that while gifted artists can do interesting things with it, this is far from true for less experienced and talented users), and (3) they do not support case-based "memories" of great designs (thereby limiting support for design by modification). These shortcomings are addressed by other efforts in our joint research work.

Integrated, Domain-Oriented Design Environments

Design environments are systems that integrate the features of domain-oriented construction kits [13] and issue-based argumentation systems [2, 19]. The major strength of *domain-oriented construction kits* is their ability to allow the designer to communicate with a set of abstractions that are meaningful within some domain of discourse. Their weaknesses are that they provide no computational support for analyzing, commenting, and critiquing designs. Issue-based argumentation systems, on the other hand, allow designers to analyze and record the issues relevant to a given design domain, but by themselves they provide no tools for the actual creation of a new artifact. Moreover, without the presence of an artifact as the focus of argumentation, they are unable to contextualize discussion to the design task at hand [9].

Integrated design environments overcome these limitations. They are based on a design methodology that integrates construction and argumentation [9] (or, in Schön's terminology, they support "reflection-in-action" [22]). This integration is made possible by the presence of *critics* [8] that analyze an artifact under construction, signal breakdown situations, and provide entry points to the space of relevant argumentation directly relevant to construction situations.

Weaknesses of Integrated, Domain-Oriented Design Environments

While design environments have proven to be a powerful concept in a large number of domains [10], they themselves are not free of their own problems. Their main shortcomings reside in (1) their limited expressiveness by providing inadequate support for design tasks not foreseen by the creator of the design environment [12, 15], and (2) while they are strong in supporting the tradition of a professional design discipline, they fall short in transcending the limits of envisioned activities [5].

Programmable Design Environments

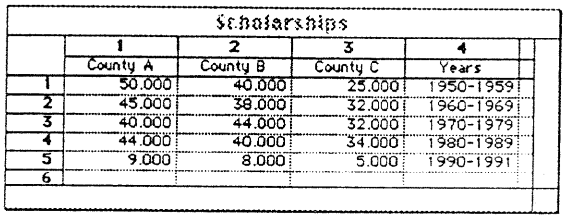
Having arrived at a point of our "design—assessment/evaluation—redesign" cycle where we have designed and identified the strengths and weaknesses of programmable applications and integrated design environments, we have articulated a conceptual framework to integrate these two approaches within programmable design environments. The goals that we want to achieve with PDEs are to create computational environments that are simultaneously expressive, supportive, and adaptable. The "theory" (i.e., knowledge is tacit, knowledge evolves, knowledge in the world interacts with knowledge in the head, etc.) behind PDEs is based on design methodologies as articulated in [3, 20, 22]. The next section illustrates the evolution of our conceptual framework in the context of a specific scenario.

THE ROAD TO PDES (2): A SERIES OF PROGRESSIVELY ELABORATED SCENARIOS

To illustrate the utility and expressive range of programmable design environments (relative to the various component paradigms described in the previous section), we consider a sequence of "progressively elaborated scenarios" drawn from the domain of graphic arts [24]. Imagine that a professional designer wishes to graph the number of scholarships awarded in several local counties by decade[1]; the following sequence shows how the same task might be undertaken within a hierarchy of "application styles" based on the taxonomy of the previous section. (It should be mentioned that although this sequence of scenarios is hypothetical—chosen for its brevity and its usefulness in illustrating all aspects of our conceptual framework—nonetheless all the strengths and weaknesses of the various paradigms have been observed in real use situations.)

Direct Manipulation

In a "pure" direct manipulation application, the designer might begin by typing in the values that she wishes to graph in tabular form, as shown in Figure 3. The column entries in this case are the number of scholarships awarded in each of three counties for the decades 1950-1959, 1960-1969, and so forth; the lone exception to this rule is represented by the fifth (and final) entry in each column, which indicates the number of scholarships in the given county not for an entire decade but for the two years 1990-1991.



Scholarships

	1	2	3	4
	County A	County B	County C	Years
1	50.000	40.000	25.000	1950-1959
2	45.000	38.000	32.000	1960-1969
3	40.000	44.000	32.000	1970-1979
4	44.000	40.000	34.000	1980-1989
5	9.000	8.000	5.000	1990-1991
6				

Figure 3: A table of data entered for the sample project.

Having entered the values in the table, the designer now selects from among the chart styles provided by the application. She selects a "line graph" format and proceeds to plot the (numeric) values of the first three columns against the (alphabetic) values entered for the horizontal-axis divisions. Chart labels and keys are added directly by typing into the appropriate locations; the dimensions of the chart are adjusted by mouse; and the resulting chart is shown in Figure 4.

This scenario illustrates several of direct manipulation's strengths, as well as some unfortunate weaknesses. On the positive side, entry of data is relatively straightforward (the designer need merely enter the items in a table); likewise, the dimensions and labels of the chart are easily

[1]This example is modeled after a graph depicted in Tufte [24], p. 60.

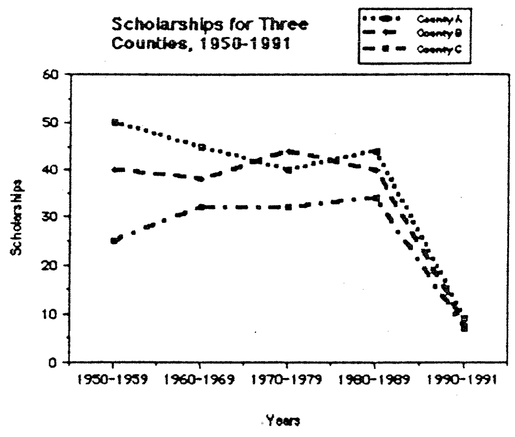


Figure 4: A sample graph created via direct manipulation.

specified. On the other hand, there are some interesting problems—missed or unconsidered opportunities—lurking just beneath the surface of this scenario. For example, the dimensions of the chart could only be chosen by eye (using the mouse to specify the size of the desired chart rectangle); thus, the designer could not scale the rectangle size to some value based explicitly on the data itself (e.g., the designer could not specify that she wants the vertical axis to extend only to the maximum of the data points). Only a limited number of graph types are available; if the designer decided, for example, that she wished to graph the scholarship-by-decade values as narrow vertical boxes ranging from the minimum to the maximum value for the decade, the option might well not be available to her. More complicated types of graph—depicting scholarships against a background of maps of the various counties, say—would likewise be out of the question. The mathematical tools available for examining the data would inevitably be limited in this "pure" direct manipulation interface (as they are in commercial applications); perhaps the designer could find the average or standard deviation of a set of data (by selecting the appropriate data-examination tools from a menu), but if she then wished to find the correlation between the number of scholarships awarded per year in County A and the number for County B, she might well be unable to do so.[2]

Perhaps most important, the resulting graph is itself problematic: because the final interval on the x-axis represents only two years (as opposed to an entire decade), the graph appears to depict a recent precipitous decline in scholarships for all three counties. (Indeed, only a careful examination of the graph would show that in fact the number of scholarships awarded has in fact increased on an average per-year basis for two of the three counties.) The choice of hatching patterns to distinguish the lines for the three separate counties is likewise controversial (though here the issue is less clear-cut); gray shading might be preferable, as this would avoid the presence of jarring "Moire effects" in the graph (cf. [24]).

[2]The typical (and less-than-optimal) decision for a professional designer under these circumstances would be to export the data to an entirely different application geared toward statistical computation, and perform the desired calculations in that separate application.

A General-Purpose Programming Environment
As an alternative to the "pure" direct manipulation scenario of the previous paragraphs, we could instead provide our designer with a general-purpose programming environment—for the purposes of illustration, this might be a Scheme environment [6] (though we note in passing that the major issues are unaffected by the choice of language).

In this scenario, the designer might construct for herself a library of procedures for creating charts and graphs of various kinds; and the data set of interest would most likely be provided to the system as a Scheme list (or, perhaps more plausibly, read in from an external database or file):

```
(define *scholarship-data-set*
  (list
   '( (County A)
      (1950 3) (1951 4) (1952 3) ....)
   '( (County B)
      (1950 3) (1951 2) (1952 2) ....)
   (etc.)))
```

In comparison to the direct manipulation scenario, the use of a programming environment presents major advantages in expressiveness [7]. *If* the designer is an accomplished programmer, she can create virtually any new type of graph imaginable; she can tailor the graph presentation so that it depends in some algorithmic fashion on the data set itself (e.g., having the program automatically label all outlying data points); and she can write customized statistical procedures to do sophisticated analysis of her data.

On the other hand, the advantages of this new scenario are predicated upon both the energy and the programming sophistication of the designer: it is assumed that she will have (and spend) the time to create a large personalized library of graphical-design procedures. It should also be recalled that the direct manipulation system did render a variety of tasks—such as choosing fonts and (when desired) "eyeballing" graph dimensions—particularly easy. Performing all these tasks via programming, while possible, is tedious.

Moreover, there are some problems with the direct manipulation scenario that have not been addressed by the introduction of programming. The same poor graph produced in Figure 4 might easily be produced by the designer/programmer as well; after all, the system provides no supporting tools for making decisions within the particular domain of graphical design. A programming language—despite its expressive range—still fails the designer on a number of important dimensions.

Domain-Oriented Construction Kits

Rather than provide the designer with either a direct-manipulation tool or a general-purpose programming environment, we might provide a "graph-construction kit" for the designer's use [13]. Such a system would have many of the same features as the direct-manipulation program—presumably, graphs would still be created by selection among a palette of predefined types—but in addition this environment might be augmented by a rich catalog of sample graphs that could be used as the basis for the designer's own work. Thus, rather than simply choose to produce a line graph from her initial data set, the designer might begin in this environment by browsing within the catalog for a graph that appears similar to the kind that she wishes to create, as shown in Figure 5.

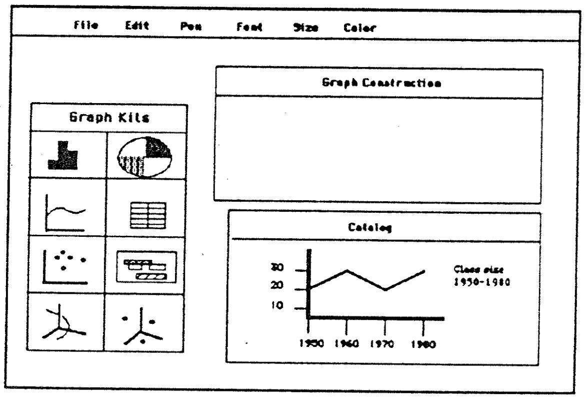


Figure 5: A view of an information-display construction kit. The designer has located a graph (within the catalog) similar to the one she wishes to construct.

The use of a construction kit with an integrated catalog alleviates some of the problems of the direct manipulation system, but fails to solve others. On the one hand, the catalog does provide illustrations of (presumably) exemplary work; thus, the designer might derive some implicit support (or advice) from the graphs included in the catalog. (For instance, if the line graphs included in the catalog distinguished between lines via gray shading or coloring rather than via hatching patterns, the designer might use these graphs as templates for her own work and thereby avoid the "Moire-pattern" effects mentioned earlier.) On the other hand, the catalog as envisioned here provides little in the way of *explicit* design rationale [9]; thus the designer might fail to consider those decisions that went into the construction of the sample graphs. (For example, the issue of "non-uniform intervals" on the x-axis might go unnoticed.) Moreover, the overall lack of expressiveness and power which caused difficulty in the direct manipulation environment is still a factor even in this more elaborate system; the strengths of the programming environment have not been introduced into the construction kit.

Argumentation System

Employing a hypermedia-based argumentation system [2, 19] would alleviate some of the difficulties to which we have alluded thus far. We might imagine the designer using such a system to explore design rationales for a variety of graphs; she might, for instance, explore an argumentation base centering on the proper representation of line graphs. Issues involving non-uniform axis intervals would very likely be addressed in this setting, forestalling errors in the designer's eventual creation.

Using an argumentation system in isolation, however, enforces an uncomfortable fragmentation in the designer's work: the implicit expectation is that she will spend a portion of her time in actual design activity and a (separate) portion of her time using the argumentation base to reflect upon design. A more realistic scenario, based on the work of Schön [22] and Ehn [5], is that design and reflection are closely interwoven. Though the features of an argumentation system are plausibly useful, they are much more compelling when embedded within a system that supports a more seamless transition between reflection and action.

Programmable Applications

A programmable application [7] for creating information displays would be aimed at integrating the characteristics of direct manipulation and programming environments (as summarized earlier in this section). Such an application could conceivably overcome some of the individual flaws of those two paradigms: the designer could now specify many typical characteristics of a graph by hand, but she could simultaneously write programs that, over time, would collectively embody an ever-growing personalized vocabulary of increasingly complex techniques for data visualization.

While alleviating many of the problems evinced separately by direct manipulation and programming, a "standard" programmable application as envisioned here nevertheless fails the designer in other ways. The advantages provided by a browsable catalog of examples, or by tools for representing design rationale (as described in the previous two subsections) are still missing; and thus the resulting application, while expressive and powerful, still fails to support the designer in learning (either about the domain or the application itself) and thereby expanding her domain knowledge or her repertoire of skills within the application [11].

Integrated Design Environments

Rather than construct a programmable application, we could pursue a different route of integration, combining the features of construction kits and argumentation systems. The result in this case would be a design environment for information displays: such an environment provides an interesting contrast to the programmable application of the previous section.

Here, the emphasis is on providing a supportive and learnable environment for the designer. Going beyond the elements of the construction kit shown in Figure 5, this environment—modelled after the example of Janus [8, 9, 12] in the domain of kitchen design—will include critics for the domain of information displays. Among these could be procedures for (1) monitoring the designer's creations-in-progress with an eye toward spotting the overuse of hatching patterns, and (2) querying the designer about the structure and semantics of her graph and acting as "agents" in finding appropriate catalog examples to use as templates. Moreover, this design environment should include an *argumentative hypermedia* system. The critics make use of elements of this system (when a critic interacts with the user, it can display representations of arguments that are relevant to the critic's particular function); similarly, the elements of this hypermedia system act as indices to relevant catalog examples [9].

Much as in the case of programmable applications, the construction of design environments alleviates some of the problems described in previous sections while leaving others untouched. Although the proposed environment is certainly rich in tools that help the designer learn and explore the domain of information displays, the absence of a programming environment places stringent (if not immediately felt) limits on her expressive range. These limits may first become apparent when the designer attempts some unforeseen task of the type described earlier in the discussion of programming environments; or she may find that programmability is desirable in working with the added subsystems of the design environment itself (e.g., she may wish to write programs that create new critics or that explore the catalog with powerful search mechanisms).

Programmable Design Environments

Our succession of scenarios has thus led us to one further act of integration—between the programmable applications and the design environments of the previous two subsections. In Figure 6, we depict a screen-view of a programmable design environment for information displays.

Using the application depicted in Figure 6, we may now pursue the original scenario (in which our hypothetical designer wishes to graph scholarships in several counties by decade). The designer might begin as she did with the direct manipulation system (or construction kit): namely, by entering data elements in a table. She now selects a type of graph to use (here, a line graph) and, as a first cut, creates a graph much like the one shown earlier in Figure 4. Having created this graph, one of the critics alerts the

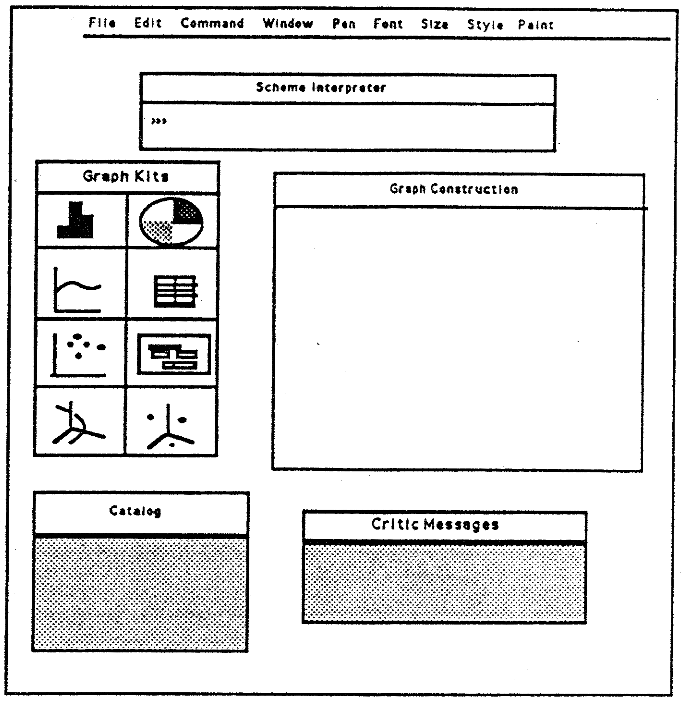
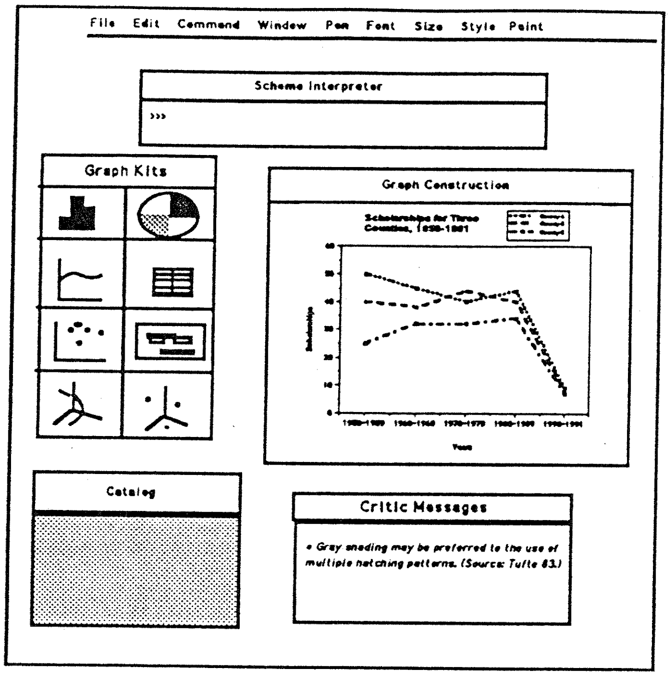


Figure 6: A screen view of a programmable design environment for graphic arts and information displays. The environment includes a "graphics-enriched" Scheme interpreter (top); a construction kit for a variety of "standard" graph types (middle left); a window for graph construction (middle right); a browsable catalog linked to an argumentation component (bottom left); and a collection of software "critics" for graphic design (which communicate with the user through messages and prompts in the window at bottom right).

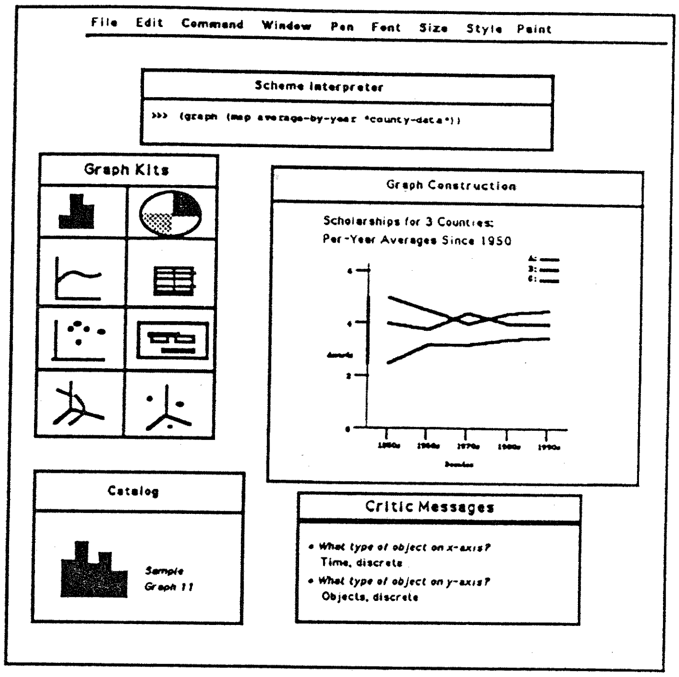
designer to the possibility that she is overusing hatching patterns, and suggests the use of gray shading instead (as shown in Figure 7).

The designer now asks the system to illustrate the critic's argumentation with a catalog entry similar to the graph that she has just created. In doing so, the system engages in a simple dialogue with the designer, asking for some information about the semantics of the graph she is creating (whether, e.g., the graph depicts numbers of discrete objects changing over time). The system finds a graph (here, a bar chart) with semantics similar to that of the designer's project, as shown in Figure 8.



• Gray shading may be preferred to the use of multiple hatching patterns. (Source: Tufte 83.)

Figure 7: The designer has used the construction kit at left to create a first draft of a graph (middle right) similar to the one shown in Figure 4. A critic message (bottom right) suggests that the newly-created graph may include an overuse of multiple hatching patterns.


File Edit Command Window Pen Font Size Style Paint
Scheme Interpreter
Graph Kits
Graph Construction
Scholarships for 3 Counties:
Per-Year Averages Since 1950
Catalog
Sample Graph 11
Critic Messages

```
>>> (graph (map average-by-year *county-data*))
```

• What type of object on x-axis?
Time, discrete
• What type of object on y-axis?
Objects, discrete

Figure 8: The designer asks the system (in the dialogue at bottom right) to locate a catalog graph similar to her own (bottom left). Using this graph as a starting point, she redoes her own graph (middle right), using a brief Scheme program to take averages of her data (top window).

In examining the argumentation surrounding this entry, the designer notes that the graph's creator took account of the fact that the time-intervals depicted on the x-axis were chosen to be constant (thus, decade measures for this graph were chosen from 1952-1961, 1962-1971, and so forth). Our designer notes that her own graph employs variable time-intervals on the x-axis and decides that rather than adopt the earlier graph-creator's solution, she will simply alter her graph to show per-year averages for each of her column entries. She writes simple programs in the Scheme interpreter (1) to take averages for each entry, and (2) to display the vertical axis in variable widths, with a thicker line depicting the range between the maximum and minimum values of all data plotted. The final graph is shown in Figure 8.

In this scenario, the designer was able to make use of a variety of techniques: (1) creating a first-draft graph via direct manipulation; (2) responding to critics by exploring argumentation; (3) obtaining an illustration of the argumentation with catalog examples; (4) reflecting on her creation by comparison with decisions made by earlier graph-designers; and (5) performing creative customization of her graph with the aid of a domain-enriched programming language.

ASSESSMENT AND FUTURE WORK

The integration of different paradigms is more than throwing a number of different things together. In our case it was driven by several cycles of "design—assessment/evaluation—redesign". Assessments of the earlier paradigms revealed their strengths and weaknesses and led us to new paradigms retaining the strengths and addressing the weaknesses at each stage. Beyond assessing prototypes, our integration efforts were theoretically founded by taking the conceptual frameworks of others (e.g., Schön's "reflection-in-action" [22], Ehn's "languages of doing" [5], Mackay's "co-adaptive systems" [18], Henderson/Kyng's "design in use" [16], etc.) into account by integrating them in our own emerging conceptual framework. PDEs themselves are raising many new interesting issues which we will investigate carefully in the future. They will be instances of high-functionality systems with their own problems requiring extensive support for (1) making them useful and usable, (2) contextualizing information to the task at hand, and (3) learning on demand [11].

ACKNOWLEDGMENTS

The authors would like to thank the members of the Human-Computer Communication group at the University of Colorado who contributed to the conceptual frameworks and systems discussed in this article. The research was supported by the National Science Foundation under grants IRI-9015441 and MDR-9253425. The first author is supported in addition by a National Science Foundation Young Investigator grant (IRI-9258684).

REFERENCES

1. Boecker, H.-D., Eden, H., and Fischer, G. *Interactive Problem Solving Using Logo*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

2. Conklin, J. and Begeman, M. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems*, Vol. 6, No. 4, October 1988, pp. 303-331.

3. Cross, N. *Developments in Design Methodology*, John Wiley & Sons, New York, 1984.

4. diSessa, A. and Abelson, H. Boxer: A Reconstructible Computational Medium. *Communications of the ACM*, Vol. 29, No. 9, September 1986, pp. 859-868.

5. Ehn, P. *Work-Oriented Design of Computer Artifacts.* Almquist & Wiksell International, Stockholm, Sweden, 1988.

6. Eisenberg, M. *Programming in Scheme*. MIT Press, Cambridge, MA, 1990.

7. Eisenberg, M. Programmable Applications: Interpreter Meets Interface. Artificial Intelligence Laboratory Technical Report 1325, MIT, 1991.

8. Fischer, G., Lemke, A.C., Mastaglio, T. and Morch, A. The Role of Critiquing in Cooperative Problem Solving. *ACM Transactions on Information Systems*, Vol. 9, No. 2, 1991, pp. 123-151.

9. Fischer, G., Lemke, A.C., McCall, R. and Morch, A. Making Argumentation Serve Design. *Human Computer Interaction*, Vol. 6, No. 3-4, 1991, pp. 393-419.

10. Fischer, G., Grudin, J., Lemke, A.C., McCall, R., Ostwald, J. , Reeves, B. and Shipman, F. Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments *Human Computer Interaction*, Vol. 7, No. 3, 1992, (in press).

11. Fischer, G. and Eisenberg, M. Mastering High Functionality Systems by Supporting Learning on Demand. Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, September 1992.

12. Fischer, G. and Girgensohn, A. End-User Modifiability in Design Environments, *CHI '90 Conference Proceedings*, pp. 183-191.

13. Fischer, G., Lemke, A.C. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction*, Vol. 3, No. 3, 1988, pp. 179-222.

14. Gantt, M. and Nardi, B. A. Gardeners and Gurus: Patterns of Cooperation Among CAD Users. *CHI '92 Conference Proceedings*, pp. 107-117.

15. Girgensohn, A. End-User Modifiability in Knowledge-Based Design Environments. Technical Report, Department of Computer Science, University of Colorado CU-CS-595-92, 1992.

16. Henderson, A. and Kyng, M. There's No Place Like Home: Continuing Design in Use. In J. Greenbaum and M. Kyng (eds.), *Design at Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum, Hillsdale NJ, pp. 219-140.

17. Hutchins, E.L., Hollan, J. D. and Norman, D. A. Direct Manipulation Interfaces. In D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 87-124, ch. 5.

18. Mackay, W.E. Co-adaptive systems: Users as Innovators. *CHI '92 Basic Research Symposium*, 1992.

19. McCall, R. PHI: A Conceptual Foundation for Design Hypermedia. *Design Studies*, Vol. 12, No. 1, 1991, pp. 30-41.

20. Norman, D. A. *Things That Make Us Smart.* Addison-Wesley Publishing Company, Reading, MA. (Expected publication, early 1993.)

21. Papert, S. *Mindstorms: Children, Computers and Powerful Ideas.* Basic Books, New York, 1980.

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

23. Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, Vol. 16, No. 8, August 1983, pp. 57-69.

24. Tufte, E. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire CT 1983.