

1991

 **University of Colorado at Boulder**

Center for LifeLong Learning and Design (L3D)

Department of Computer Science

---

ECOT 717 Engineering Center  
Campus Box 430  
Boulder, Colorado 80309-0430  
(303) 492-1592, FAX: (303) 492-2844

**The Role of Visual Representations in  
Understanding Software**

Heinz-Dieter Böcker  
Institut für Informatik  
Universität Stuttgart  
Stuttgart, Germany

Gerhard Fischer and Helga Nieper-Lemke  
University of Colorado  
Department of Computer Science  
Campus 430  
Boulder, CO 80309-0430 USA

"Artificial Intelligence and Software Engineering", D.Partridge (Ed.)  
Ablex Publishing Co., pp. 273-290.

# The Role of Visual Representations in Understanding Software

Heinz-Dieter Böcker  
 Institut für Informatik  
 Universität Stuttgart  
 Herdweg 51  
 7000 Stuttgart 1  
 FRG

Gerhard Fischer and Helga Nieper-Lemke  
 Department of Computer Science  
 Campus Box 430  
 University of Colorado  
 Boulder, CO 80309-0430  
 USA

## Abstract

The way a problem is represented strongly affects our ability to understand and solve it. Visual representations are especially important because the human visual system is such a powerful way of processing information. However few existing systems try to take advantage of these insights. In pursuit of the long-range goal of constructing a *software oscilloscope* that makes the invisible visible, we have constructed system components which automatically generate graphical representations of complex structures, illustrate the control flow of complex programs, and support visualization techniques in object-oriented environments. Our tools are used in a variety of contexts: in programming environments, as components in intelligent support systems, and in human-computer interaction in general. Visual representation alone, however, is not enough; the designer of visualization tools must take into account the semantics of graphical symbols and the user's need to limit visualization to the *relevant* facts and relations.

## 1. Introduction

The way a problem is represented strongly affects whether we can understand and solve it. Simon [Simon 81] argues that solving a problem simply means representing it so as to make the solution transparent. He argues that this is especially true for mathematics; mathematical derivations can be viewed as changes in representation, making evident what was previously true but obscure.

Believing that the limits of our thoughts are all too often identical with the limits of our capacity to imagine and visualize, we have developed over the last several years a rich variety of visualization tools to make using computers a more rewarding and less error-prone experience. Our goal is to build software components that take advantage of the power of the human visual system to provide insight and understanding, instead of relying only on verification methods. Being especially interested in ill-structured problems, we have found (like all other researchers investigating design problems empirically) that the recommendation "think more clearly" is not good enough; overwhelming evidence shows that there is an urgent need for better tools because humans have a bounded rationality. Verification systems [Millo, Lipton, Perlis 79] that end up with the result "correct" or "incorrect" contribute little to our understanding of a problem. Moreover, verification procedures rely on an exact specification of a problem, whereas the crucial activity in solving ill-structured

problems is to come up with this specification in the first place. In the following sections, we will present first steps towards a *software oscilloscope* that helps users understand the behavior of complex artifacts. The components of this software oscilloscope all exploit the powerful human visual system.

With workstations and bitmap displays becoming widely available, many people now recognize the importance and power of visualization tools. The growing number of publications reflects the increasing interest in visualization and visual programming (e.g., [Smith 77; Gould, Finzer 84; Brown 88; Reiser et al. 88; Ambras, O'Day 88; Ingalls et al. 88]). A good survey is contained in [Computer 85].

One of the most important applications of visualization tools is software design, because the gradual shift from designing software from scratch to creating software through redesign and reuse [Fischer 87a] requires the programmer first to understand the software in its existing form.

## 2. Insight and Understanding versus Verification

Figure 2-1 summarizes two different views of the crucial issues in computer science [Newell, Simon 76; Fischer, Boecker 83].

	View 1	View 2
<b>Computer science is:</b>	a formal, mathematical discipline	an experimental discipline
<b>Main tools for the development of systems are:</b>	formal specification techniques	rapid prototyping, experimental programming
<b>Basic challenge is:</b>	think more clearly	develop better tools (because humans have a bounded rationality)
<b>Programming methodology:</b>	do not write programs which cannot be verified before they are written	design is an error correcting process

Figure 2-1: Two Opposing Views of the Crucial Issues in Computer Science

Obviously, the appropriate view depends on the application area. For all the areas in which we have been interested (e.g., artificial intelligence, cognitive science, human-computer communication, use of computers for learning and instruction), the second view is most adequate, and it is this perspective that has governed our construction of the tools described in Section 3.

Rather simple experiments may be used to demonstrate the importance of graphical representations in problem solving:

- **The Rope around the Earth** (see [Fischer 79]): A rope is tied around the earth at the equator (assuming that the surface of the earth is smooth). If we extend this rope by one yard and form a concentric circle around the earth, will the difference between the earth and the rope be big enough that a mouse can get through? Almost everybody's intuitive answer to this problem is "no." Simple mathematics easily proves that the resulting difference is *independent* of the size of the surrounded object and definitely big enough so that a mouse can get through. This proof *verifies* the result, but it provides *no insight and understanding*. How do we make people believe the proof, i.e., understand the solution? One possibility is to ask them to consider the following problem: A rope is lying on the ground between Boulder and Denver (or any other two cities that are close to each other). Can we lift it up by ten inches without increasing the length of the rope? This thought experiment indicates the relationship between radius and curvature.
- **Number Scrabble and Tic-Tac-Toe** (see [Simon 81], page 151, for details) are two isomorphic versions of the same game. Yet subjects perform much better at Tic-Tac-Toe than at Number Scrabble. It is possible that Tic-Tac-Toe is easier to play because it is more visually oriented.
- **The Design of a Roulette Table**: Teaching high school students problem solving with LOGO, we asked them to simulate a roulette table with slots 0 to 18. Given was a random number generator that returned a number between 0 and 9. Most students felt quite happy with the solution "sum of random and random." Not until they plotted the results in a graph did they discover that their roulette table did not give a uniform distribution. The visual representation of the results did not show to them how to produce a fair roulette table, but it did uncover the incorrect solution in an obvious way.

Our experiments with these and similar problems suggest that the right kind of representation can provide insight and understanding into a problem. One of the advantages of visual representations over their formal, propositional counterparts is that visual representations facilitate direct observation of important properties, which is usually cheaper (in terms of the computations involved) than deduction.

### 3. Visualization Tools

Myers proposes that visualization-oriented programming tools may be classified into two main categories: "program visualization" and "visual programming" [Myers 86]. In our opinion, the distinction between these two types of system is much less clear cut than Myers suggests. The systems described in the following sections are all in-between cases. For instance, KAESTLE is a program visualization tool as well as a visual programming tool because it allows the user to edit the structures displayed (see Section 3.1). The distinction becomes totally blurred for object-oriented systems in which the graphical objects of the visualization have direct counterparts within the programming formalism (see Section 3.3).

#### 3.1 Visualization of Data Structures: KAESTLE

The most important data structure of LISP is the list. KAESTLE [Boecker, Nieper 85] automatically generates graphical representations of list structures and allows the user to edit them directly with a pointing device (Figure 3-1).

KAESTLE helps the LISP beginner to understand certain aspects of the programming language that are difficult to explain otherwise (e.g., the difference between copying and destructive functions; Figure 3-2). More experienced LISP programmers use it heavily to display and explore data structures that are difficult to represent symbolically, namely circular and reentrant structures (see `kaestle-window-1` in Figure 3-1). KAESTLE, as part of a programming environment, can be used to design, debug, document, and understand LISP programs or data structures.

**Planning the Spatial Layout of List Structures.** To generate a graphical representation of a list structure, it is necessary to find a position on the screen for each element belonging to the structure (for more details see [Boecker, Nieper 85]). A *fully* automated layout planning algorithm faces the following problems:

- List structures may be complex networks. A very good but time-consuming planning algorithm is not helpful in an interactive system.
- Often, there is not enough space to display the entire structure. How can it be decided which parts of the structure to omit?
- The semantics of the list structure (i.e., its logical structure) should be taken into account.

Therefore our basic approach is to build *cooperative* systems [Fischer 88] for the human and the computer. In KAESTLE, the computer uses a simple planning algorithm: It doesn't pay attention to arrows crossing boxes or other arrows, and the display is truncated at the right and lower border of the display area. After this "first draft" of the graphical representation is generated by the computer, users can move or delete parts of the display, or display additional substructures, until they arrive at a "nice" output that shows the parts of the structure they want to see.

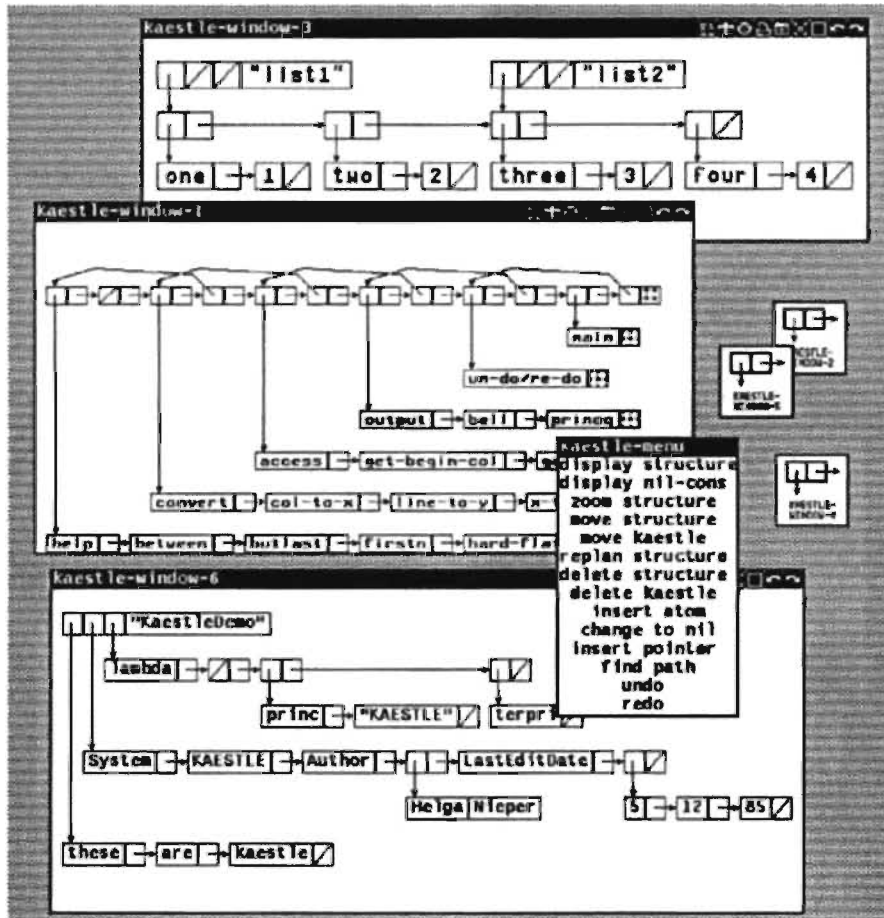


Figure 3-1: KAESTLE, a Graphical Editor for List Structures

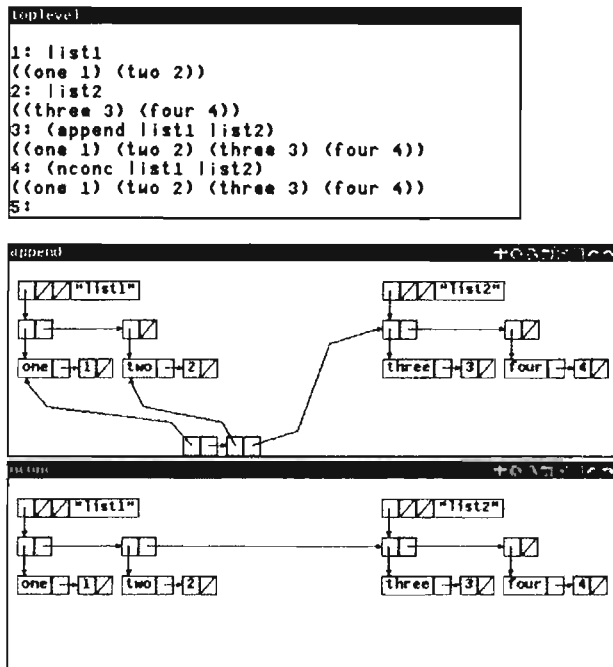


Figure 3-2: The Difference Between `append` and `nconc`

This figure illustrates the effects of the copying function `append` and the destructive function `nconc`. The normal textual representation displayed in the `toplevel` window reveals no difference between the results of these two functions.

**Functionality of the System.** KAESTLE provides the following operations, among others:

- Generating a graphical representation: displaying multiple independent structures at any desired position.
- Changing the graphical representation: displaying substructures that are truncated in the current display, deleting substructures from the display, moving substructures on the screen.
- Changing the underlying list structure: inserting atoms or pointers in the graphical representation immediately changes the underlying list structure.
- Undo and redo mechanisms.

**Dynamic Aspects: What Happens to the Structures?** KAESTLE is used not only to display and edit static structures but also to monitor running programs. The standard LISP trace package can be used for this purpose by updating the graphical representation whenever an “interesting” function is entered or left. The trace facility can also be used to generate a sequence of snapshots of a data structure while the program is running (see Figure 3-3, which illustrates a recursive, destructive algorithm that reverses a list).

```
(def my-nreverse
  (lambda (l)
    (cond ((dtpr (cdr l))
           (progl (my-nreverse (cdr l))
                  (rplacd (cdr l) l)
                  (rplacd l nil)))
          (t l))))
```



Figure 3-3: A Sequence of Snapshots of a Data Structure

**A Case Study: Self-Organizing Linear Lists.** The following case study shows how KAESTLE is used to debug a LISP function. KAESTLE allows tinkering with data structures and supports primitive forms of “programming by example.” Users may test algorithms on specific examples and have KAESTLE keep a record of what happens to the data structures. Visualization tools like this reduce the conceptual distance between the symbols and primitives of a LISP function and the manipulated data structure. They turn a cons-cell into an object that can be easily manipulated.

The LISP function to be debugged in our example reorganizes a linear list by pulling an element to the front of the list whenever the element is accessed, in order to speed up later operations. The list is implemented as an association list in LISP. Figure 3-4 shows how the `self-org-assq` access function should reorganize the association list `a-list`.

```
1: a-list
((Francois . Paris) (Maggie . London)
 (Helmut . Bonn) (Ronald . Washington))

2: (self-org-assq 'Helmut a-list)
(Helmut . Bonn)

3: a-list
((Helmut . Bonn) (Francois . Paris)
 (Maggie . London) (Ronald . Washington))
```

Figure 3-4: Self-Organizing Linear Lists

In implementing the algorithm to reorganize this list, one of the authors of this paper had defined a buggy `self-org-assq` access function that for some unknown reason chopped off the last element of the list; Figure 3-5 shows what happened.

```
1: a-list
((Francois . Paris) (Maggie . London)
 (Helmut . Bonn) (Ronald . Washington))

2: (self-org-assq 'Helmut a-list)
(Helmut . Bonn)

3: a-list
((Helmut . Bonn) (Francois . Paris)
 (Maggie . London))
```

Figure 3-5: A Buggy Implementation

The question was: What happened to the last element? Without visualization tools, it would have been necessary to undertake the tedious and error-prone task of debugging that function by essentially running a simulation of it, drawing cons-cells with pencil and paper and making heavy use of an eraser to redirect pointers. With the help of KAESTLE, the bug was easily discovered. Figure 3-6 shows what happened internally: The `cdr` of the last cell was erroneously made to point to itself. With this clue, the code was easy to fix. No other tool of the LISP programming environment would have made diagnosis this simple.

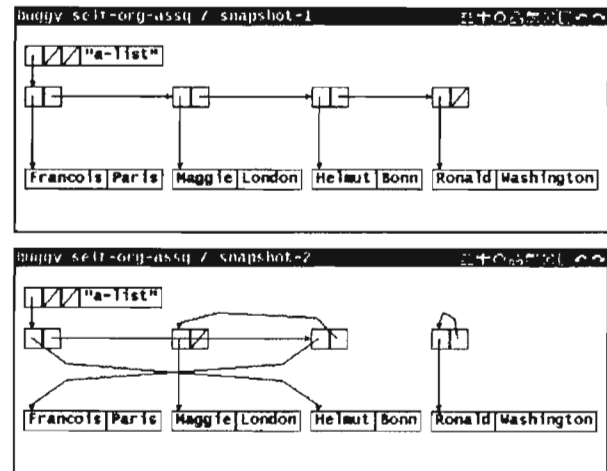


Figure 3-6: How (Ronald . Washington) Was Isolated

### 3.2 Visualization of Control Structures: FOOSCAPE

A program composed of a large set of usually rather simple functions may be appropriately described as a *network of functions* that mutually call each other. FOOSCAPE displays functions as ellipses that are connected by arrows (cf. Figure 3-7). The tool is meant primarily to give the user a first, rough overview of some piece of software. It is especially useful for languages that do not allow lexical nesting of function definitions.

The planning of the layout (placement of the ellipses and arrows) is done automatically (see [Boecker, Nieper 85] for more details). However, because the solution sometimes is not “beautiful,” FOOSCAPE allows the user to modify the

layout interactively by moving the ellipses around or by altering the set of functions included in the display. The tool's interaction style is similar to that of KAESTLE.

Traditional techniques for monitoring the dynamic behavior of programs (e.g., breakpoints, dumps) suffer from the fact that they capture just one state of the data and too often generate huge amounts of data. FOOSCAPE tries to avoid these disadvantages and to preserve the dynamics of the processes being monitored. Being able to see a program run gives one a grasp of detail that is hard to obtain in any other way.

FOOSCAPE not only displays the static calling structure of a program; it can also be readily used to display the program's dynamic behavior. The basic mechanism for accomplishing this is provided by the standard LISP trace package. Figure 3-7 shows a snapshot of an animated FOOSCAPE. A function name is highlighted, that is, flips from white to black, whenever the function is active.



Figure 3-7: An Animated FOOSCAPE

The impression given by a "running" FOOSCAPE bears some resemblance to the control panels of (outdated) computer systems: You can tell from the pattern of lights what the system is doing. We also added sound to the FOOSCAPE tool: Each of the functions is assigned two specific tones that are played when a function is entered and left, respectively. Initial experience with this experimental version seems to confirm that the human audio system is even more capable of monitoring sequences over time than the human visual system: *As long as the program plays this Bach style music everything is ok.*

The usefulness of the tool depends on its appropriate use: The programmer has to exercise care in selecting the functions to be included in the FOOSCAPE. If the granularity is too fine (the functions included are too primitive) only a

flickering screen will be seen, whereas if the granularity is too coarse hardly any dynamic behavior will be observed. In order to control the granularity, the user can temporarily exclude certain functions from being traced (see the functions shaded with a gray raster-pattern in Figure 3-7).

### 3.3 Visualization in Object-Oriented Formalisms

Object-oriented programming formalisms are well suited to visual representation. The objects of an object-oriented language like SMALLTALK may naturally be mapped into graphical objects to be displayed on the screen. If the user can directly manipulate these graphical objects with the help of a pointing and dragging device like a mouse, the distinction between "visual programming" and "program visualization" quickly fades away.

#### 3.3.1 Zoo, a Knowledge Acquisition Tool

ZOO [Riekert 87] provides the user with a graphical interface for objects of the OBJTALK programming language [Rathke, Lemke 85]. It allows knowledge engineers to inspect object-oriented knowledge bases and to modify and augment a knowledge base by directly manipulating screen objects. The graphic representation provides two kinds of graphic primitives: icons and labeled arrows (Figure 3-8). Icons are used to represent objects; the graphic symbol visualizes the class membership of the object. Knowledge can be modeled as a network of icons as nodes and labeled arrows as links. The user creates classes and instances by copying and modifying the icons of existing classes and instances. They may then be linked together with other objects to add knowledge to the knowledge base under construction. Deleting objects from the knowledge base is done in similar ways.

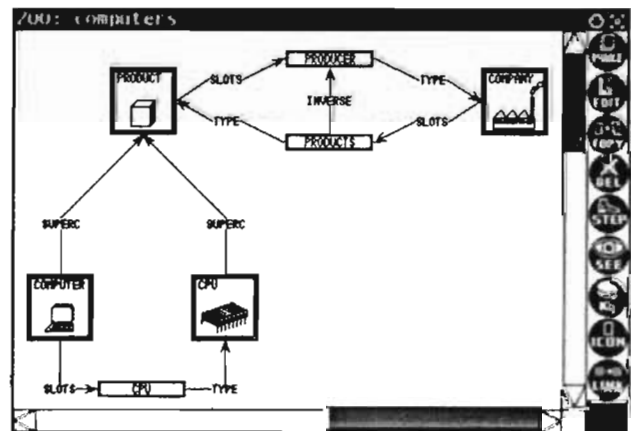


Figure 3-8: The Knowledge Editor ZOO

This figure displays the knowledge that computers and CPUs are both products, products are produced by companies, and companies produce products (the inverse relationship, which is generated automatically).

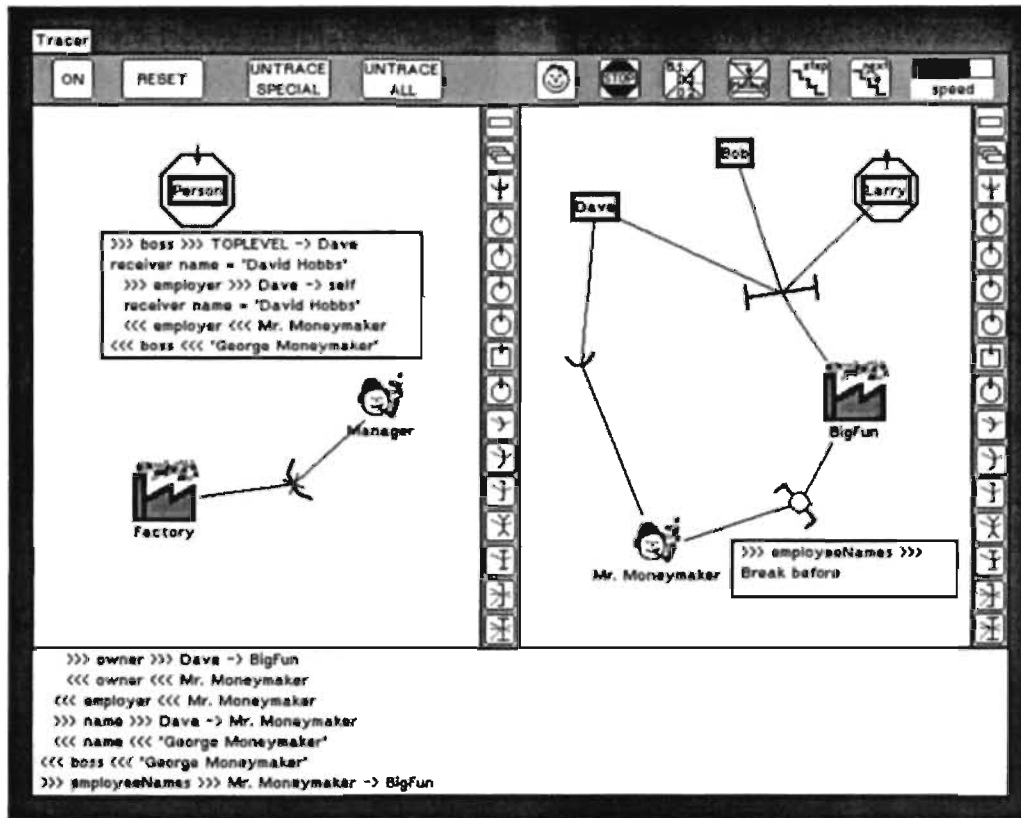


Figure 3-9: TRACK, a Trace Construction Kit

### 3.3.2 TRACK, a Trace Construction Kit

TRACK [Herczeg 89] extends the basic idea underlying FOOSCAPE into the domain of object-oriented programming. It is implemented in SMALLTALK [Goldberg, Robson 83] and fully integrated into the SMALLTALK programming environment.

TRACK is used to trace messages sent between the objects of a SMALLTALK program. The users first place icons representing objects on the screen. They may then select "hurdles" from a menu and place them between the objects. The type of hurdle and its position determine the messages to be traced and the objects involved in the trace (Figure 3-9). The user may specify constraints in terms of the message's pattern, the sender, the receiver, the class of the method that handles a message, etc. Different hurdle icons represent the different types of traces (e.g., round icons represent more general ones, icons resembling square brackets represent more specific ones). Big circles, squares, and octagons may be used to surround specific objects with "walls" and thus specify object-specific traces. Also, breakpoints may be specified together with the hurdles. In the simplest case, all messages traveling between two specific objects in either direction would be traced. The flow of the message from one object to another is indicated by small circles traveling in real time between objects (in Figure 3-9, a message is just crossing the hurdle set up between "Mr. Moneymaker" and the "BigFun" company). More detailed information about the messages sent may be viewed in text windows connected to the hurdles.

Traces may be set up for classes (left pane in Figure 3-9) or individual instances (right pane in Figure 3-9). The speed of the tracing is adjustable; programs may also be run in stepping mode. The lower part of a TRACK window dynamically displays all messages monitored by the hurdles and walls that are currently set up.

### 3.4 Visualization of Directed Graphs: TRISTAN

Unlike the systems described above, which are specific to one application (e.g., to display and edit LISP data structures), TRISTAN is a *generic* tool for the display of directed graphs [Nieper-Lemke 88].

TRISTAN, together with domain-specific knowledge (e.g., how to compute the parents and children of a node, what it means to insert or remove a link, or how a node is called in the application), makes a domain-specific graph editor. An application programmer who wants to implement a new TRISTAN application uses TRIKIT, a form-oriented design environment [Fischer, Lemke 88a; Fischer, Lemke 88b]. With the help of TRIKIT, the application programmer sets and adjusts the parameters of TRISTAN and thus specifies the interface between TRISTAN and the application.

TRISTAN has been used for several applications, including inheritance hierarchies of object-oriented formalisms (Figure 3-10), file hierarchies, the newsgroup hierarchy of the USENET news system, rules of a rule base, and state graphs.

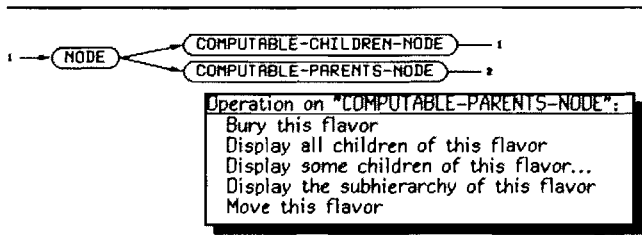


Figure 3-10: TRISTAN: Display of an Inheritance Hierarchy

This figure shows part of the inheritance hierarchy of flavors that are used to implement TRISTAN itself. The little numbers indicate how many children (or parents) a node has that are currently invisible. Also shown is the context-sensitive menu that appears on a flavor node.

**Functionality of TRISTAN.** The following operations are available in TRISTAN:

- displaying an arbitrary subset of nodes and the corresponding links (e.g., single nodes, all children of a node, a subhierarchy of a node);
- changing the graphical representation (e.g., displaying additional nodes or making currently displayed nodes invisible, moving nodes, replanning the layout of subhierarchies);
- changing the underlying structure (e.g., creating or deleting nodes or links);
- highlighting of nodes;<sup>1</sup>
- defining mouse actions, which get activated when a node is selected with the mouse (e.g., to display a node in more detail).

## 4. Visualization and Beyond

### 4.1 Usage of Visualization Tools

The visualization tools described in this paper have been used regularly by a large group of researchers and students. This use has triggered new ideas for creating additional tools of the same kind and applying them as building blocks in larger applications. For widespread use, it is critically important that these tools be tightly integrated and easily accessible within the general programming environment. Nothing is a better indication of the usefulness of a tool than that people start using it without being forced to (e.g., on the job) or asked to (e.g., in a psychological experiment). KAESTLE has been used within the LISP-CRITIC [Fischer 87b], a tutorial system that criticizes the users' programs and offers immediate explanation and justification for the criticisms, using actual data taken from the current work context.

<sup>1</sup>This operation could be used to implement the animation feature of FOOSCAPE in a system that uses TRISTAN to display the calling hierarchy of a program.

Our visualization tools may be used to complement other tools, like video disks, in a natural way. The main advantage of our tools is that they free the designer of the explanation facilities from foreseeing all conceivable future situations. Explanations can be generated on the fly and do not have to be precompiled and stored for later use. Integrating these tools with models of the user [Fischer, Lemke, Schwab 85] allows advice and information to be given only when they are relevant for the actual situation.

### 4.2 Lessons Learned

One of the most striking lessons that we have learned in implementing the various kinds of visualization tools relates to the automatic planning of graph layouts. Brandenburg [Brandenburg 89] has shown for several aesthetics parameters (like area covered, width of graph, number of crossings) that the production of nice drawings of graphs and even trees becomes computationally intractable; in general, they are NP-complete. By loosening the requirements on niceness and by taking into account application-related dependencies of graphs one can usually find algorithms that produce *some* solution within an acceptable time. But even harder problems arise because people do not easily agree on what the properties of a *nice* layout would be. The semantics of the structures to be displayed sometimes require alternative representations that cannot be deduced from the syntax of the structures. For instance, the spatial layout of a list structure that is understood by the user as an implementation of a higher-level data structure (e.g., a torus's surface topology) has to reflect the semantics of this higher-level structure. To produce high-quality visualizations, therefore, human and computer must share an understanding of domain-oriented concepts [Fischer, Lemke 88b].

One question still to be answered is whether these ideas and methods will scale up to "real" problems involving hundreds or thousands instead of the tens of objects used in our examples (e.g., representing cons-cells or LISP functions). What additional techniques will we have to invent to cope with these large spaces?

### 4.3 Human Problem-Domain Communication

Our experience has shown that successful visualizations do not guarantee a successful use of the computer. Most existing visualization tools are purely graphic and lack semantic qualities (e.g., a note in a music editor is a mere bitmap). Clearly, visualization-based systems can be greatly improved if they are augmented by deep representations of the knowledge underlying the relevant problem domain. Systems that *combine* knowledge representation and visualization techniques can achieve much better communication between humans and computers.

Most computer users (e.g., office workers, physicists, musicians, user interface designers, kitchen designers) are experts in some specific problem domain. They are not interested in learning the "languages of the computer;" they simply want to *use* the computer to solve problems and accomplish tasks. To shape the computer into a truly usable and useful medium, we have to make it invisible and let users work directly on their problems and their tasks.



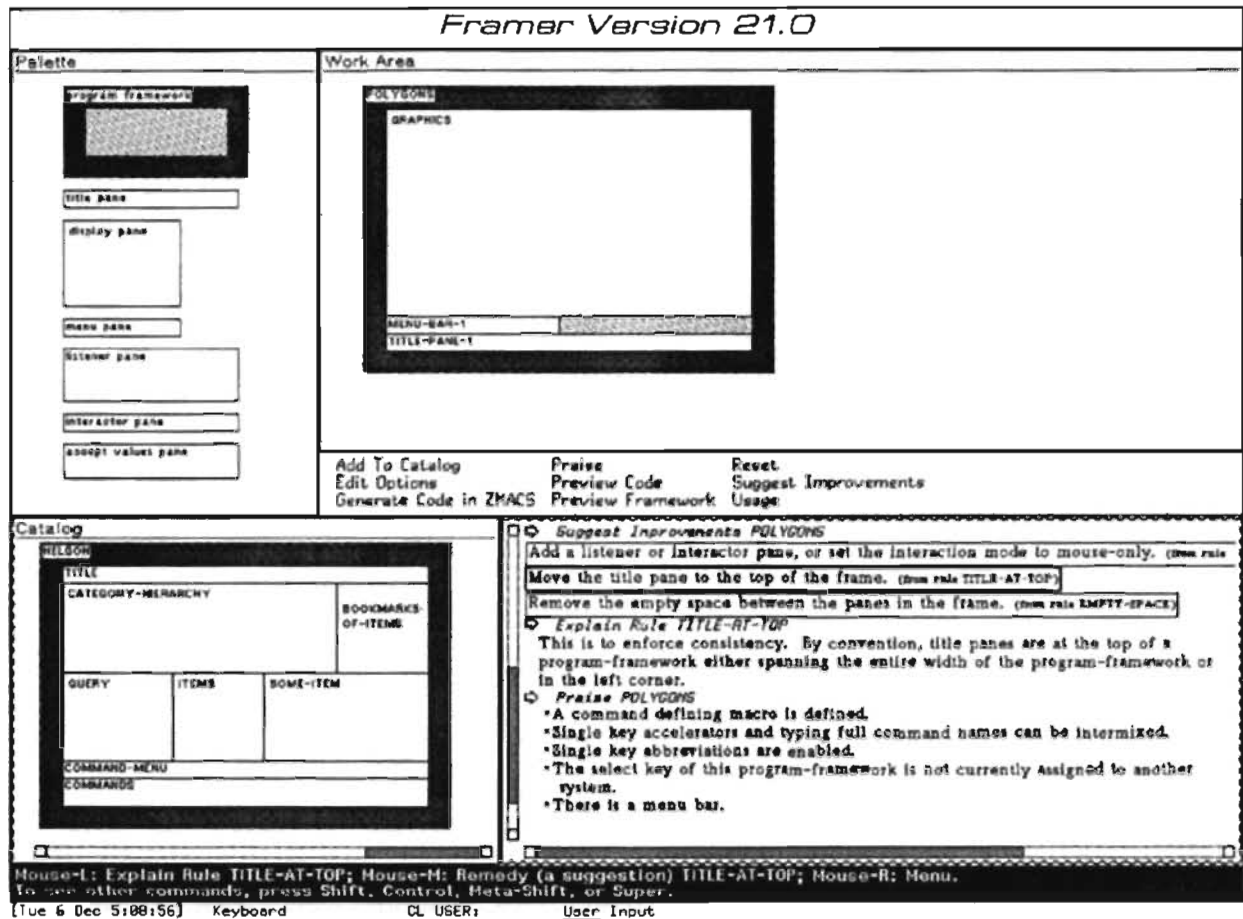


Figure 4-1: FRAMER, a Design Environment for Window-Based Interfaces

We must “teach” the computer the languages of experts by endowing it with the abstractions of different application domains. This reduces the transformation distance between the domain expert’s description of the task and its representation as a computer program.

Human *problem-domain* communication [Fischer, Lemke 88b] provides a new level of quality in human-computer communication, because the important abstract operations and objects of a given application area are built directly into the computing environment. The user can thus operate with personally meaningful abstractions. In most cases we do not want to eliminate the semantics of a problem domain by reducing the information to formulas in first-order logic or to general graphs. We have built a series of systems [Boecker, Mahling 88; Boecker, Herczeg, Herczeg 89; Fischer, Lemke 88b; Fischer, Morch 88; Fischer, McCall, Morch 89] that demonstrate this basic idea. The idea of “visual programming” has to be reformulated under the human problem-domain communication paradigm. Visual programming in this sense will be less purely a matter of manipulating icons or symbols and can be more appropriately understood within the broader context of domain-dependent design activities.

**FRAMER, an Example for Human Problem-Domain Communication.** FRAMER is a design environment for constructing window-based user interfaces (Figure 4-1). Design environments reduce the amount of knowledge designers have to acquire before they can do useful work. FRAMER permits users to design their own user interfaces without writing code and thus supports human problem-domain communication. It offers the user a palette of domain-oriented building blocks that can be directly manipulated to create a new design. The visual interaction style is specifically appropriate for a problem domain in which visual objects are designed from visual parts.

In addition to serving as an application-oriented construction kit, FRAMER has a small rule base incorporating design knowledge about relevant aspects of window-based user interfaces. The **Praise** command tells a user what is good about a design, whereas the **Suggest Improvements** command criticizes it. The **Explain** option gives the user a rationale for the suggested improvement. The **Catalog** contains a number of prototypical designs that can be praised, critiqued, or brought into the work area to be modified and used as a starting point for redesign. Such prototypical solutions that can be changed and refined through redesign are an important source of possibilities for designers. After having created an interesting design, users can add it to the **Catalog**.

## 5. Conclusions

The commercial success of systems taking advantage of rather simple visualization techniques (e.g., spreadsheet programs) indicates that visually based software has great potential for making computer systems attractive to people who have previously been alienated and scared by their formal nature and their nontransparency. Our experience with the visualization tools described above has shown that they can make computers understandable and transparent for all kinds of users.

Many interesting problems remain to be solved in this area. Not the least of these problems is to build visualization tools for a large variety of applications and eventually come up with a toolkit so that they can be easily constructed. In many situations, however, it is not good enough to make the invisible visible [Boecker, Nieper 85]. What is required is techniques that help the user make the *relevant* facts and relations visible, e.g., intelligent summarizers and filtering techniques.

The paradigm of human problem-domain communication allows us to focus on the semantics of graphical symbols and the design aspects of the problem-solving activities carried out with them. Graphical objects have to be more than nice pictures on the screen. They need to be reactive and responsive and have to be backed up by extensive knowledge representation mechanisms that turn them into virtual world objects. Visualization is often a necessary, but not a sufficient condition for understanding.

## Acknowledgments

The authors would like to thank their colleagues Jürgen Herczeg, who developed TRACK, Andreas Lemke, who developed FRAMER, and Wolf-Fritz Rickert, who developed ZOO. Nancy Mann helped us edit the paper. The research was partially supported by the Deutsche Forschungsgemeinschaft (DFG), by the Bundesministerium für Forschung und Technologie (BMFT), and by grant No. IRI-8722792 from the National Science Foundation, grant No. MDA903-86-C0143 from the Army Research Institute, and grant No. 0487.12.0389B from U S WEST Advanced Technologies.

## References

- [Ambras, O'Day 88]  
J. Ambras, V. O'Day, *MicroScope: A Knowledge-Based Programming Environment*, IEEE Software, May 1988, pp. 50-58.
- [Boecker, Herczeg, Herczeg 89]  
H.-D. Boecker, J. Herczeg, M. Herczeg, *ELAB -- An Electronics Laboratory*, Proceedings of the Fourth Conference on Artificial Intelligence and Education (Amsterdam), May 1989.
- [Boecker, Mahling 88]  
H.-D. Boecker, A. Mahling, *What's in a Note?*, Proceedings of the International Computer Music Conference '88, GIMIK, Cologne, FRG, 1988, pp. 166-174.
- [Boecker, Nieper 85]  
H.-D. Boecker, H. Nieper, *Making the Invisible Visible: Tools for Exploratory Programming*, Proceedings of the First Pan Pacific Computer Conference, Australian Computer Society, Melbourne, Australia, September 1985, pp. 563-579.
- [Brandenburg 89]  
F.J. Brandenburg, *Nice Drawings of Graphs are Computationally Hard*, in P. Gorny, M.J. Tauber (eds.), *Visualization in Human-Computer Interaction*, Springer-Verlag, Berlin - Heidelberg - New York, Lecture Notes in Computer Science 1989.
- [Brown 88]  
M.H. Brown, *Perspectives on Algorithm Animation*, Human Factors in Computing Systems, CHI'88 Conference Proceedings (Washington, D.C.), ACM, New York, May 1988, pp. 33-38.
- [Computer 85]  
*Computer*, Special Issue on Visual Programming, Vol. 18, No. 8, IEEE Computer Society, August 1985.
- [Fischer 79]  
G. Fischer, *Multiple Representations*, MMK-Memo, Institut fuer Informatik, Universitaet Stuttgart, 1979.
- [Fischer 87a]  
G. Fischer, *Cognitive View of Reuse and Redesign*, IEEE Software, Special Issue on Reusability, Vol. 4, No. 4, July 1987, pp. 60-72.
- [Fischer 87b]  
G. Fischer, *A Critic for LISP*, Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), J. McDermott (ed.), Morgan Kaufmann Publishers, Los Altos, CA, August 1987, pp. 177-184.
- [Fischer 88]  
G. Fischer, *Cooperative Problem Solving Systems*, Proceedings of the 1st Simposium Internacional de Inteligencia Artificial (Monterrey, Mexico), October 1988, pp. 127-132.
- [Fischer, Boecker 83]  
G. Fischer, H.-D. Boecker, *The Nature of Design Processes and how Computer Systems can Support them*, Integrated Interactive Computing Systems, Proceedings of the European Conference on Integrated Interactive Computer Systems (ECICS 82), P. Degano, E. Sandewall (eds.), North-Holland, 1983, pp. 73-88.
- [Fischer, Lemke 88a]  
G. Fischer, A.C. Lemke, *Constrained Design Processes: Steps Towards Convivial Computing*, in R. Guindon (ed.), *Cognitive Science and its Application for Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, pp. 1-58, Ch. 1.
- [Fischer, Lemke 88b]  
G. Fischer, A.C. Lemke, *Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication*, Human-Computer Interaction, Vol. 3, No. 3, 1988, pp. 179-222.

- [Fischer, Lemke, Schwab 85]  
G. Fischer, A.C. Lemke, T. Schwab, *Knowledge-Based Help Systems*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), ACM, New York, April 1985, pp. 161-167.
- [Fischer, McCall, Morch 89]  
G. Fischer, R. McCall, A. Morch, *Design Environments for Constructive and Argumentative Design*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, April 1989.
- [Fischer, Morch 88]  
G. Fischer, A. Morch, *CRACK: A Critiquing Approach to Cooperative Kitchen Design*, Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), ACM, New York, June 1988, pp. 176-185.
- [Goldberg, Robson 83]  
A. Goldberg, D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley Publishing Company, Reading, MA, 1983.
- [Gould, Finzer 84]  
L. Gould, W. Finzer, *Programming by Rehearsal*, Technical Report SCL-84-1, Xerox Palo Alto Research Center, May 1984.
- [Herczeg 89]  
J. Herczeg, *TRACK -- Ein Werkzeug zur Realisierung eines graphisch visuellen Traces fuer Smalltalk-80*, Diplomarbeit, Institut fuer Informatik, Universitaet Stuttgart, 1989.
- [Ingalls et al. 88]  
D.H.H. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, K. Doyle, *Fabrik: A Visual Programming Environment*, OOPSLA'88 Conference Proceedings, Special Issue of ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988.
- [Millo, Lipton, Perlis 79]  
R.A. De Millo, R.J. Lipton, A.J. Perlis, *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, Vol. 22, No. 5, May 1979, pp. 271-280.
- [Myers 86]  
B.A. Myers, *Visual Programming, Programming by Example, and Program Visualization: A Taxonomy*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 59-66.
- [Newell, Simon 76]  
A. Newell, H.A. Simon, *Computer Science as an Empirical Inquiry: Symbols and Search*, Communications of the ACM, Vol. 19, No. 3, 1976, pp. 113-136.
- [Nieper-Lemke 88]  
H. Nieper-Lemke, *TRISTAN, ein generischer Editor fuer gerichtete Graphen*, in R. Gunzenhaeuser, H.-D. Boecker (eds.), *Prototypen benutzergerechter Computersysteme*, Walter de Gruyter, Berlin - New York, 1988, pp. 243-257, Ch. XIV.
- [Rathke, Lemke 85]  
C. Rathke, A.C. Lemke, *ObjTalk Primer*, Technical Report CU-CS-290-85, Department of Computer Science, University of Colorado, Boulder, CO, February 1985.
- [Reiser et al. 88]  
B.J. Reiser, P. Friedmann, J. Gevins, D.Y. Kimberg, M. Ranney, A. Romero, *A Graphical Programming Language Interface for an Intelligent Lisp Tutor*, Human Factors in Computing Systems, CHI'88 Conference Proceedings (Washington, DC), ACM, New York, May 1988, pp. 39-44.
- [Riekert 87]  
W.-F. Riekert, *The ZOO Metasystem: A Direct-Manipulation Interface to Object-Oriented Knowledge Bases*, ECOOP'87, European Conference on Object-Oriented Programming (Paris, France), Springer-Verlag, Berlin - Heidelberg - New York, June 1987, pp. 132-139.
- [Simon 81]  
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.
- [Smith 77]  
D.C. Smith, *Pygmalion, A Computer Program to Model and Stimulate Creative Thought*, Birkhaeuser Verlag, Basel, 1977.