Construction and Design Kits: Human Problem-Domain Communication

Gerhard Fischer and Andreas C. Lemke

CU-CS-366-87 June 1987

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-1502

Construction and Design Kits: Steps Toward Human Problem-Domain Communication

Gerhard Fischer and Andreas C. Lemke

Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder, Colorado 80309-0430

Contribution to: Human-Computer Interaction --
published by Lawrence Erlbaum Associates
(editor: Ruven Brooks)

Abstract: Our goal is to build cooperative computer systems to augment human intelligence. In these systems the communication between the user and the computer plays a crucial role. Knowledge-based systems make special demands on human-computer communication, but they also provide new unique opportunities to enhance this communication. To provide the user with the appropriate level of control and a better understanding, we have to replace human-computer communication with *human problem-domain communication*, which allows users to concentrate on the problems of their domain and to ignore the fact that they are using a computer tool.

Construction and design Kits are system components that represent steps towards human problem-domain communication. A construction kit is a set of building blocks that models a problem domain. The building blocks define a design space (the set of all possible designs that can be created by combining these blocks). Design kits go beyond construction kits in that they bring to bear general knowledge about design (e.g., which meaningful artifacts can be constructed, how and which blocks can be combined with each other) that is useful for the designer. Prototypical examples of these systems (especially in the area of user interface design) are described in detail and the feasibility of this approach is evaluated.

Table of Contents

List of Figures

1. Introduction

In today's world, the construction of complex systems is limited not only by what is technologically feasible but also by what is desirable and manageable by humans. Theories and methodologies from cognitive science are an important source of knowledge about and insight into the construction of *user-centered systems* [Norman, Draper 86; Fischer, Kintsch 86]. Research in cognitive science should help us both to understand our own human nature and to conduct our practical affairs. It should provide the scientific basis for cognitive engineering, that is, for a cognitive approach to the design of human-computer systems.

Many aspects of human-computer systems have not kept pace with the dramatic progress in hardware technology. In particular, it is difficult for even the expert - let alone the novice and occasional user - to take advantage of the available computational power *to use the computer for a purpose chosen by him/herself*. Most computer users feel that computer systems are unfriendly, uncooperative and that it takes too much time and effort to get something done. They feel dependent on specialists, and they notice that "software is not soft"; that is, the behavior of a system cannot be changed without reprogramming it substantially. Casual users find themselves in a situation similar to instrument flying: they need relearning lessons after not using the system for a while. We claim that systems fail primarily because their communication capabilities are insufficient.

In this paper we will first characterize research efforts in human-computer communication by articulating some problems. We will describe different approaches and determine the specific challenges and unique opportunities that knowledge-based systems create for human-computer communication. *Human problem-domain communication* will be seen as a major step towards increasing the conviviality of computer systems. To achieve conviviality [Fischer, Lemke 87], construction kits and design kits (seen as instances of intelligent support systems) are needed. We will describe in detail some of the design kits that we have constructed over the last few years. Finally, we will evaluate our experience and discuss potential future work.

2. Human-Computer Communication

We believe that the term *user interface* should be replaced by *human-computer communication* because communication between humans and computers requires more than tacking another layer of software onto a computer system.

We are concerned with a new class of computer systems that support *cooperative problem solving* and provide *advice*, *criticism*, and *explanation*. In these systems the boundaries between the user interface portion and the application system become much less clear than in traditional systems. Knowledge-based systems are the most promising approach to improve human-computer communication because successful communication is based on knowledge structures common to both human and computer [Fischer 83a].

In the following sections we look at some of the general problems in human-computer communication and at some approaches that have been used to solve them.

2.1 Problems of Human-Computer Communication

Designers of communication processes between humans and computers are challenged by a large number of requirements. They must:

- help break the *complexity barrier* (e.g., by supporting dynamic unfolding and user- and task-specific filters);
- help break the *utility barrier*, defined as the ratio of value to effort expended (either by increasing the value of a system or by decreasing the effort needed to learn and use it);
- *give control to users* when they need or desire it; do things automatically when users do not want to be bothered;
- promote *human problem-domain communication*; mirror the abstractions of the application domain, thereby reducing the transformation distance between task descriptions by the domain expert and their representations as computer programs; *mental models* must be developed in the context of human problem-domain communication.
- take advantage of modern hardware and basic software capabilities; for example, use screens as a two-dimensional world that can be edited (*direct manipulation*);
- *support active exploration* (e.g., undo and redo operations).

Knowledge-based systems pose special problems and at the same time offer great possibilities for human-computer communication. The more intelligent and autonomous a system is, the harder it is to understand and the more important become issues of reliability, comprehensibility, and trust in the system's performance [Chambers, Nagel 85].

Many knowledge-based systems are built under the assumption that the user has a well defined problem that the system is supposed to solve. This assumption has led to strongly system-controlled advisory dialogues (e.g. in the MYCIN system [Buchanan, Shortliffe 84]) that provide little help in *problem definition*. Frequently, however, users want to understand the nature of their problems incrementally and solve them in cooperation with a system [Woods 86]. This requires better communication capabilities than most systems traditionally have offered.

Empirical evidence (failed attempts to build fully automatic systems, e.g. in machine translation; see [Winograd, Flores 86]) has shown that for many domains, a symbiotic, cooperative system architecture is more adequate and promises greater success than an autonomous one. For symbiotic, cooperative systems, a human-computer interaction subsystem is an absolute necessity. It is our belief that in many ways partially autonomous systems pose greater design challenges than fully autonomous systems. The two agents have to keep each other informed about their decisions and actions, and one of the central questions is: who is in control when there is a conflict of opinion? Knowledge-based systems develop their "own will," which may be viewed as an encapsulation of their designers' will and understanding of the situation.

These observations provide the rationale for one of the major research topics for the future: *how to bring knowledge-based systems and human-computer communication together to construct systems that are useful and usable.* Knowledge-based systems have considerable amounts of domain knowledge, most of which is represented explicitly; they therefore meet the important requirement for intelligent human-computer interaction that not all information has to be communicated explicitly [Fischer 83b]. This advantage over systems that are not knowledge-based must be exploited.

To overcome some of these problems, we have developed *qualitative design criteria* for human-computer systems (for a more complete list see [Fischer 84]):

1. *The limiting resource in human processing of information is human attention and comprehension, not the quantity of information available.* Modern information and communication technologies have dramatically increased the amount of information available to individuals. This can be illustrated with an example from modern aircraft design [Chambers, Nagel 85]: there are 455 separate warnings on a Boeing 747. We need instruments that not only display but also prioritize information before presenting it to the crew to avoid an information overload.

2. *The limitations and structure of human memory must be taken into account in designing human-computer communication.* People have limited short-term memories. The way people recognize information is different from the way they recall memory structures. This distinction is relevant, for example, to judging the advantages and limitations of different interaction models, such as comparing a command-based interface to a menu-based interface. Our intelligence has become partially externalized, contained in artifacts as much as in our head: the computer is in one sense an artificial extension of our intellect, invented by humans to extend human thought processes and memory.

3. *The efficiency of human visual processing capabilities must be utilized fully.* Traditional interfaces have been one-dimensional, with a single frame on the screen usually filled with lines of text. New technologies allow us to take advantage of human visual perception through the use of multiwindow displays, color, graphics, and icons. To exploit these possibilities, we have constructed a user interface construction kit (see section 4.2 and Figure 4-2) and components of a *"software oscilloscope"* [Boecker, Fischer, Nieper 86].

These qualitative design criteria can be used to provide some global guidance for the construction of better human-computer communication possibilities and have played a crucial role in the development of the systems described in this paper. A drawback of them is that they are not prescriptive enough to indicate how we can and should proceed within the context of a specific system design.

2.2 Approaches to Human-Computer Communication

User Interface Management Systems (UIMS). UIMS [Olsen et al. 84] provide graphic primitives and tools for specifying dialogue structures (ATNs, context free languages). They are reasonable approaches to problems in which there is only a limited information exchange and a *strong* separation between user interface and application system. The kinds of problems we try to solve (e.g., building intelligent support systems like help, documentation and explanation systems) have convinced us that a strong separation between interface and application is a limiting factor. A user interface should have extensive access to the state and actions of the application system, and the user should be able to influence the behavior of the application.

Natural Language Interfaces. Because of the *asymmetry* between human and computer, the design of the interface is a problem not only of simulating human-to-human communication but of engineering alternatives in the domain of interaction-related properties [Bolt 84]. We do not have to use natural language for every application; some researchers claim that in many cases it is not the preferred mode of communication [Bates, Bobrow 84; Robertson, McCracken, Newell 81]. In natural language interfaces, the computer is the listener and the human the speaker. The listener's role is always more difficult because the problem must be understood from the speaker's description. Our work has been primarily guided by the belief that the user is more intelligent and can be directed into a particular context. This implies that the essence of user-interface design is to provide users with appropriate cues. Windows, menus, suggestions lists, forms, and so on (see Figure 4-2) provide a context that makes the machine the speaker and the human the listener, thereby allowing the user's intelligence to keep choosing the next step.

Currently there exists a *front-end fallacy* in human-computer communication. An appropriate interactive behavior is assumed to have been accomplished when some off-the-shelf natural language front end is tacked onto an existing system. Many human-computer systems, however, have to perform more sophisticated functions than answering requests for factual information; for example, they must help users formulate their problems and assist in cooperative problem solving. These tasks require more elaborated data models and knowledge representations [Williams et al. 82] and additional types of reasoning.

Rather than building systems that can analyze ever more complex sentences involving increasingly difficult semantic concepts, a main objective of natural language interface research should be to understand the processes of intention communication and recognition well enough to enable a system to participate in a natural dialogue with its user [Winograd, Flores 86]. Assuming we had a natural language interface to UNIX [Wilensky et al. 84], we probably would be unpleasantly surprised if our question *"How can I get more disc space?"* were answered by *"Type rm *"*, which deletes all files in a directory, even though this command would solve the problem as stated. The problem in human-computer interaction is not simply that communicative troubles arise that do not occur in human communication, but that when they do arise, there are not the same resources available for their detection and repair.

Direct Manipulation Interfaces. Natural language interfaces are based on a conversation metaphor, whereas direct manipulation interfaces are based on a world metaphor [Hutchins, Hollan, Norman 86]. In a conversation metaphor the user must translate intentions into expressions that the interface intermediary can understand. In a world metaphor the user can be directly engaged with the objects of the world (see Section 4.1). A real world metaphor can be exploited to make systems easier to learn and understand. However, problems occur in direct manipulation when recurring operations and processes need to be described.

Human Problem-Domain Communication. Most computer users are not interested in computers per se, but they want to use the computer to solve problems and to accomplish their 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.

3. Human Problem-Domain Communication

Human problem-domain communication provides a new level of quality in human-computer communication because it permits us to build the important abstract operations and objects of a given application area directly into the environment. This implies that the user can 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. Whenever the user of a system can directly manipulate the concepts of an application, programs become more understandable, and the distinction between programmers and non-programmers vanishes.

3.1 Modeling Problem Domains

Many large software systems are built as monolithic systems, completely implemented in a general purpose programming language. Human problem-domain communication requires to build one (or more) levels of intermediate layers of more problem-oriented building blocks (Figure 3-1). An architecture of this kind provides good support for programming methodologies based on redesign (modifying the original design) and reuse (recombining the intermediate abstractions to form a different system; for details about reuse and redesign see [Fischer, Lemke, Rathke 87]).

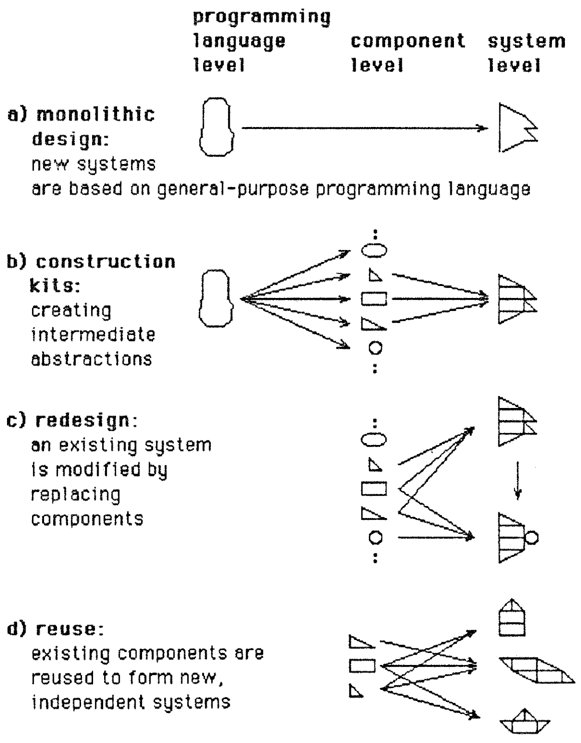

programming language level
component level
system level
a) monolithic design:
new systems are based on general-purpose programming language
b) construction kits:
creating intermediate abstractions
c) redesign:
an existing system is modified by replacing components
d) reuse:
existing components are reused to form new, independent systems

Figure 3-1: Modeling problem domains with application-oriented abstractions

Human problem-domain communication requires environments which support design methodologies whose main activity is not the generation of new, independent programs, but the integration, modification, and explanation of existing ones [Winograd 79]. Just as one relies on already established theorems in a new mathematical proof, new systems should be built as much as possible using existing parts. In order to do so, the designer must understand the functioning of these parts. An important question concerns the level of understanding necessary for successful redesign: exactly how much does the user have to understand? Our methodologies (differential programming and programming by specialization based on our object-oriented language ObjTalk [Rathke 86]) and support tools are steps in the direction of making it easier to modify an existing system than to create a new one. Inheritance is important for redesign because it enables objects that are almost like other objects to be created easily with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification by allowing information to be entered and changed in one place.

Our experience developing a construction kit for user interface design (see section 4.2) indicates that the development of the right kind of abstractions (and their embedding into inheritance hierarchies) is a difficult process which takes time and which has to proceed in an evolutionary fashion driven by the development of application systems based on these abstractions.

3.2 Intelligent Support Systems

Systems, which make an attempt to model many different problem domains will be large and complex in order to provide all the necessary abstractions (examples are the LISP machines and our WLISP programming environment; for a quantitative assessment see [Fischer, Lemke 87]). The abstractions, represented in the case of WLISP by more than 200 ObjTalk classes, comprise stable intermediate parts for development of user interfaces. The large number of classes in WLISP is, however, a mixed blessing. The advantage is that in all likelihood a building block or set of building blocks that either fits our needs or comes close to doing so already exists and has already been used and tested. The disadvantage is that they are useless unless the designer knows that they are available. Informal experiments [Fischer 87] indicate that the following problems prevent designers from successfully exploiting the potential of high functionality systems:

- designers do not know about the existence of needed objects (either building blocks or tools);
- designers do not know how to access objects;
- designers do not know when to use these objects;
- designers do not understand the results objects produce for them;
- designers cannot combine, adapt and modify an object for their specific needs.

Unless we are able to solve these problems, designers will constantly reinvent the wheel instead of taking advantage of already existing tools.

In highly complex systems, communication between humans and computers cannot be restricted to the construction of nice pictures on the screen, and the beauty of the interfaces must not overshadow the limited functionality and extensibility of some systems. The "intelligence" of a complex computer system must contribute to its ease of use. Truly intelligent and knowledgeable human communicators, such as good teachers, use a substantial part of their knowledge to explain their expertise to others. In the same way, the "intelligence" of a computer should be applied to providing effective communication. Equipping modern computer systems with more and more computational power and functionality will be of little use unless we are able to assist the user in taking advantage of them. Empirical investigations [Fischer, Lemke, Schwab 85] have shown that on the average only a small fraction of the functionality of complex systems such as UNIX, EMACS or LISP is used.

In Figure 3-2 we illustrate a system architecture that we have developed in response to our design criteria. We have constructed a number of prototypical systems of the outer ring including a documentation system [Fischer, Schneider 84], help systems [Fischer, Lemke, Schwab 85], critics [Fischer 87] and design kits.

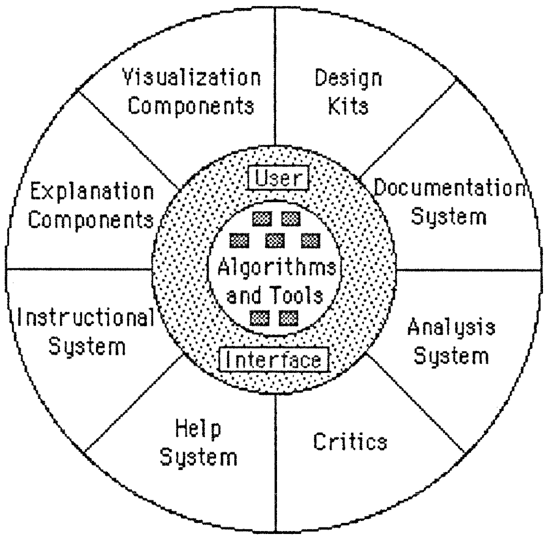


Figure 3-2: From interactive to intelligent systems

4. Construction Kits

A construction kit is a set of building blocks that models a problem domain. The building blocks define a design space, i.e., the set of all possible designs that can be created by combining these blocks. The building blocks in our systems are organized as inheritance networks in an object-oriented architecture, which provides for components on multiple levels and facilitates the extension of the set of available blocks. Simon [Simon 81] demonstrates that the evolution of a complex system proceeds much faster if stable intermediate parts exist.

We have studied and built construction kits in many domains [Fischer, Boecker 83; Fischer, Lemke, Rathke 87] to gain a deeper understanding of design processes and to increase the conviviality of computer systems through human problem-domain communication. In the following sections, we will briefly describe some examples and we will indicate how the problem of adding new elements to a construction kit can be addressed.

4.1 The PinBall and Music Construction Kits

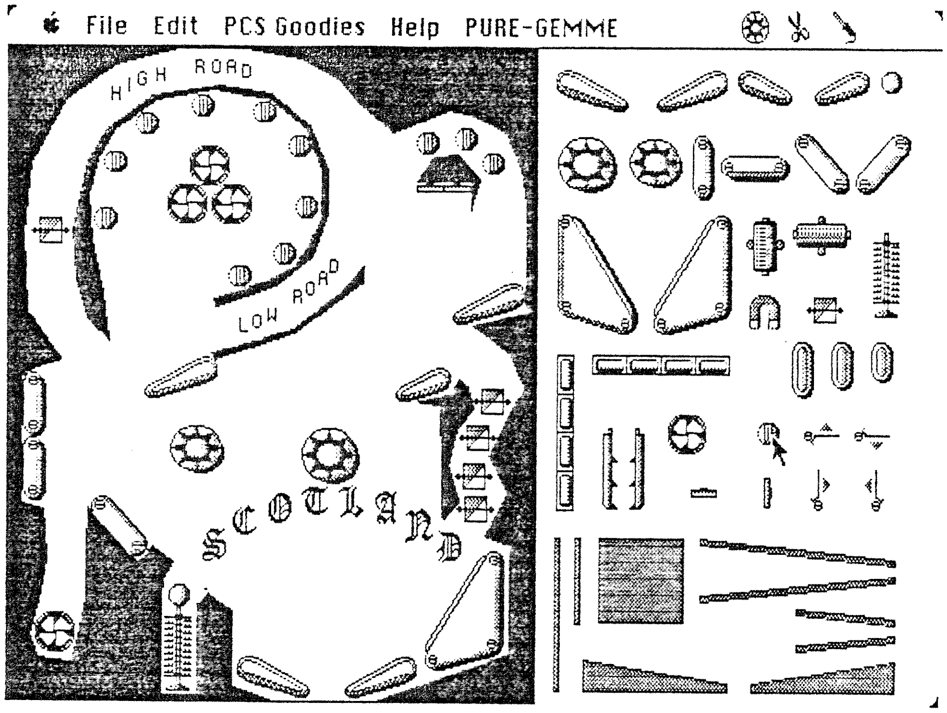
The PinBall and Music Construction Kits (two interesting programs for the MacIntosh from Electronic Arts; see Figure 4-1) provide domain-level building blocks (bumpers, flippers; staves, piano keyboard, notes, sharps, etc.) to build artifacts in the two domains of pinball machines and musical composition. Both kinds of systems reduce learning processes by exploiting the user's knowledge of the problem domain. Users can interact with the system in terms with which they are already familiar; they need not learn abstractions peculiar to a computer system.

Our empirical investigations have shown that these systems come close (within their scope) to our notion of human problem-domain communication. Users familiar with the problem domains but inexperienced with computers had few problems using these systems, whereas computer experts unfamiliar with the problem domains were unable to exploit the power of these systems. Most people considered it a very difficult (if not impossible) task to achieve the same results using only the basic Macintosh system without the construction kits.

Persons using the systems do programming, but the programming consists of constructing artifacts in the domain and not of writing statements of a programming language. Our subjects had a sense of accomplishment in using the construction kits, because they were creating their own impressive version of something that works, yet is not difficult to make.

Evaluating the Pinball and Music Construction kits as prototypical examples against our objective to enhance human problem-domain communication, we have identified the following shortcomings:

1. The two systems eliminate programming errors below the domain level, but they do not assist the user in constructing interesting and useful artifacts in the application domains. The pinball construction kit allows users to build sets in which balls get stuck in certain corners and certain devices can never be reached [Hutchins, Hollan, Norman 86]. To assist users in constructing truly interesting objects, design kits are needed;
2. The space of modeled abstractions is not extensible by the user;
3. It is unclear how generalizable the framework (consisting of spatial organization and combining functionality in simple ways between the parts, e.g., associating sound with a bumper) is to other problem domains.



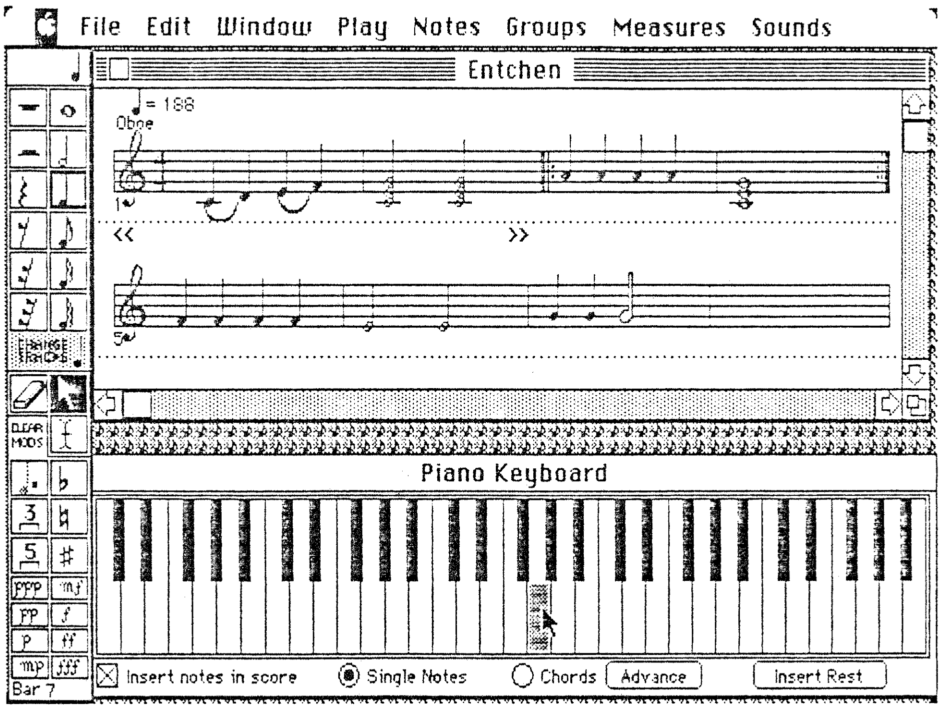
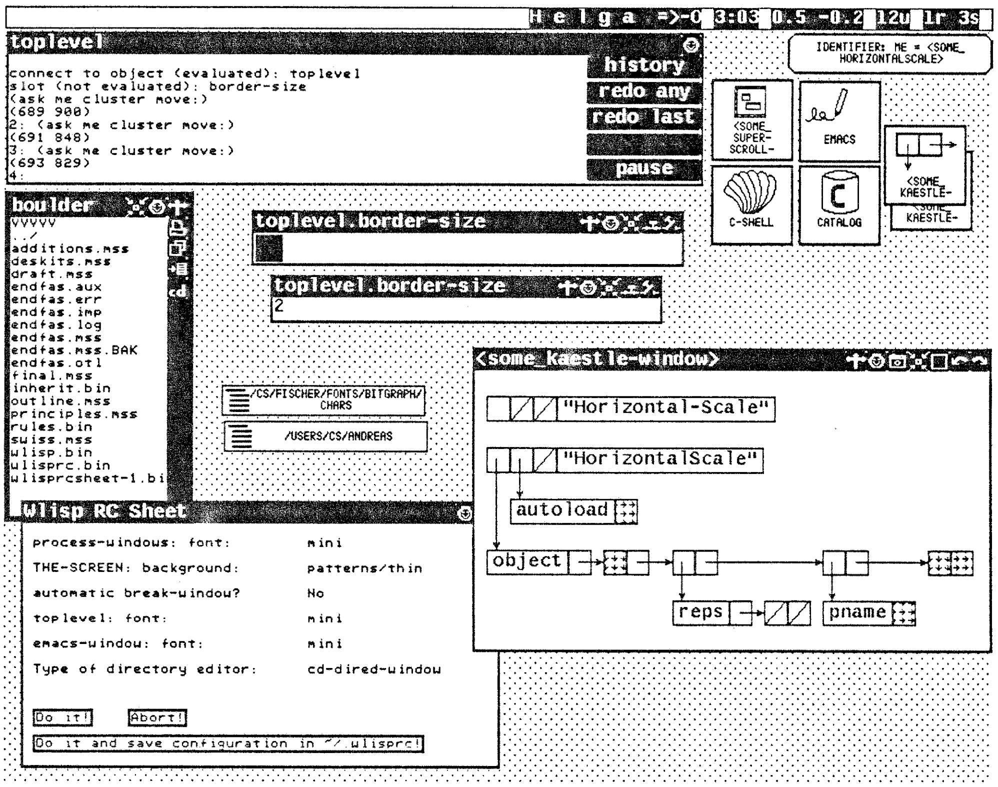


Figure 4-1: Screen images from the Pinball and Music Construction Kits

4.2 WLISP: A Construction Kit for User Interfaces

Over the last several years, we have developed WLISP [Fabian 86; Boecker, Fabian, Lemke 85], an object-oriented, knowledge-based construction kit for human-computer communication, and a large number of associated tools and intelligent support systems for exploiting this kit effectively (Figure 4-2). In contrast to pinball machine design, it was (and still is to some extent) unclear what the right abstractions in user interface design are. The WLISP system currently consists of over 200 classes representing abstractions about different kinds of windows such as super-windows, paned windows, menus, icons, gauges, etc.. The inheritance network is still changing and indicates our growing understanding about the domain of two-dimensional interfaces.



```
connect to object (evaluated): toplevel
slot (not evaluated): border-size
(ask me cluster move:)
(689 900)
2: (ask me cluster move:)
(691 848)
3: (ask me cluster move:)
(693 829)
4:
```

```
VVVVV
../
additions.mss
deskits.mss
draft.mss
endfas.aux
endfas.err
endfas.imp
endfas.log
endfas.mss
endfas.mss.BAK
endfas.otl
final.mss
inherit.bin
outline.mss
principles.mss
rules.bin
swiss.mss
wlisp.bin
wlisprc.bin
wlisprcsheet-1.bi
```

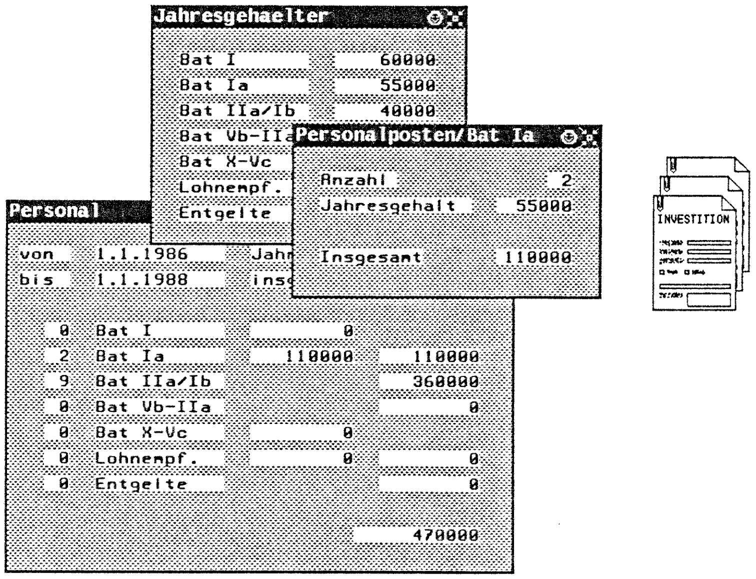
process-windows: font:	mini
THE-SCREEN: background:	patterns/thin
automatic break-window?	No
toplevel: font:	mini
emacs-window: font:	mini
Type of directory editor:	cd-dired-window

Figure 4-2: The WLISP programming environment

Using WLISP, the "human-computer communication design question" is answered by providing appropriate building blocks that suggest good designs. The object-oriented system architecture is highly flexible and enhances the reusability of many building blocks. In creating new human-computer communication capabilities the designer may use existing objects either directly or with minor modifications and can thereby rely on standard and well-tested components.

4.3 FINANZ: A Financial Planning Kit

FINANZ [Rathke 86] is an advanced financial planning system extending the spread sheet paradigm (see Figure 4-3). It differs from ordinary spread sheet programs in that the relationships among the form fields are represented by internal knowledge structure that bear knowledge about the application domain.



Jahresgehaelter

Bat I	60000
Bat Ia	55000
Bat IIa/Ib	40000
Bat Vb-IIa	
Bat X-Vc	
Lohnempf.	
Entgelte	

Personalposten/Bat Ia

Anzahl	2
Jahresgehalt	55000
Insgesamt	110000

Personal

von	1.1.1986	Jahr	
bis	1.1.1988	insg	
0	Bat I	0	
2	Bat Ia	110000	110000
9	Bat IIa/Ib		360000
0	Bat Vb-IIa		0
0	Bat X-Vc	0	
0	Lohnempf.	0	0
0	Entgelte		0
			470000

Figure 4-3: The financial planning system FINANZ

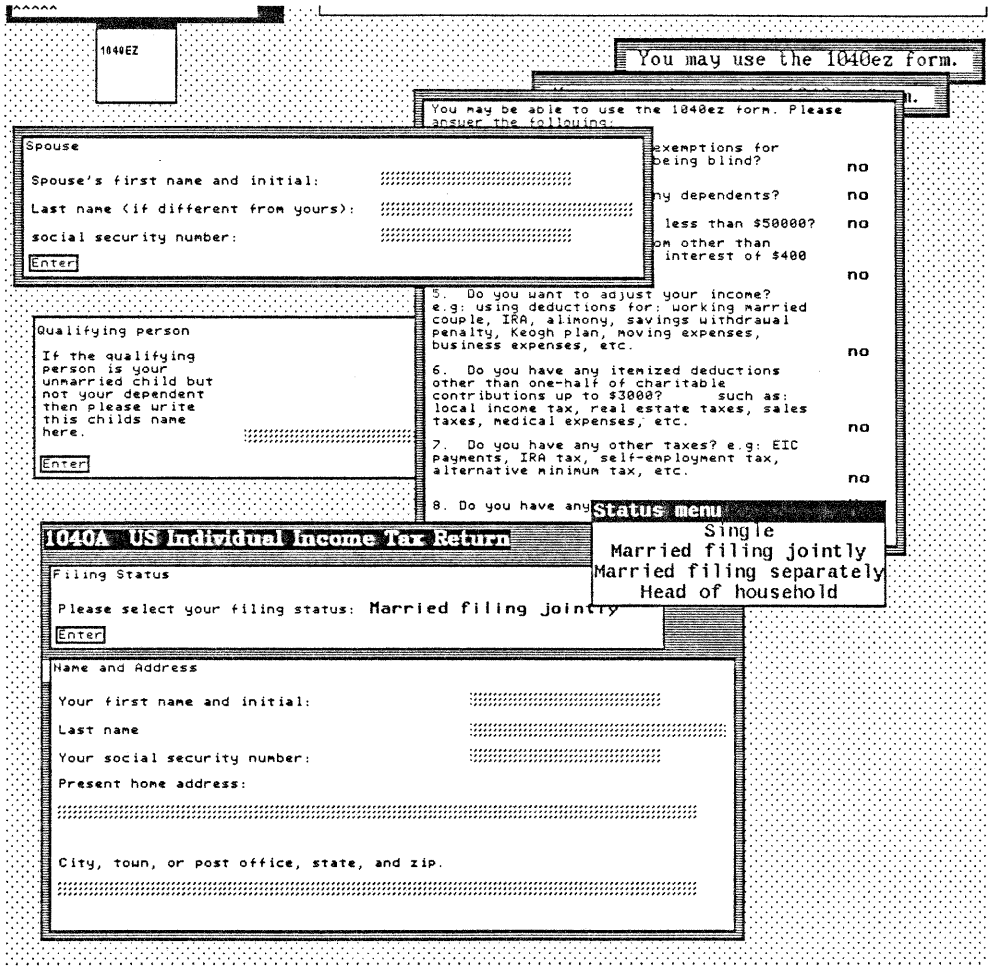
The main characteristics of FINANZ are (for details see [Rathke 86]):

- In its basic configuration it can be used as a regular spread sheet system. From there it can be gradually augmented to a knowledge-based system without loosing its basic supportive way of interaction.
- It is embedded in a window based direct manipulation environment that makes it easy to specify operations among the form fields. Multiple forms can be displayed and operated on at the same time. Operations between form fields are selected from a menu.
- The system can be augmented to incorporate knowledge about the domain to which it is applied. Relationships among the form fields are expressed by internal knowledge structures, which can be modified to serve the needs of the application.
- Internal knowledge structures are used to generate context dependent explanations on the fly. These explanations reflect the domain specific knowledge as well as the current state of the dialogue.

The capabilities of the systems are based on ObjTalk [Rathke 86]. Concepts about the application domain, the user and the dialogue are represented as active objects that communicate by message passing. Their behavior is described in classes, that form a hierarchy among which knowledge is inherited. By specifying dependencies among the form fields the user generates internal knowledge structures that not only maintain the consistency but are also used to provide context dependent help. The specification of the dependency structures requires little programming knowledge because it is done using direct manipulation techniques. Fields that take part in a new relationship are pointed at with the mouse.

4.4 WFORMS: An Electronic Forms Kit

Forms are a generally useful user interface tool. The WFORMS kit, a facility of WLISP, provides specialized abstractions for designing electronic forms (Figure 4-4). The general organizational model is a hierarchy of alternating vertical and horizontal structuring elements called blocks and lines. The primitive elements are different types of fields that support the display of text and the input of values by selecting alternatives with the mouse, pop-up-menu selection, or typing.



You may use the 1040ez form.

You may be able to use the 1040ez form. Please answer the following:

exemptions for being blind? no

ny dependents? no

less than $50000? no

om other than interest of $400 no

5. Do you want to adjust your income? e.g: using deductions for: working married couple, IRA, alimony, savings withdrawal penalty, Keogh plan, moving expenses, business expenses, etc. no

6. Do you have any itemized deductions other than one-half of charitable contributions up to $3000? such as: local income tax, real estate taxes, sales taxes, medical expenses, etc. no

7. Do you have any other taxes? e.g: EIC payments, IRA tax, self-employment tax, alternative minimum tax, etc. no

8. Do you have any

Spouse

Spouse's first name and initial:

Last name (if different from yours):

social security number:

Qualifying person

If the qualifying person is your unmarried child but not your dependent then please write this childs name here.

1040A US Individual Income Tax Return

Filing Status

Please select your filing status: Married filing jointly

Name and Address

Your first name and initial:

Last name

Your social security number:

Present home address:

City, town, or post office, state, and zip.

Figure 4-4: The WFORMS electronic forms kit

4.5 ZOO: Graphical Support to Construct New Elements for a Construction Kit

ZOO [Riekert 86], implemented in WLISP and ObjTalk, provides graphical support (Figure 4-5) for constructing new domain-dependent abstractions without being forced to go down to the ObjTalk or even the LISP level (a possibility which we indicated was missing from the Pinball and Music construction kits). It is a menu-driven system in which design support is given through the organization of menus similar to the suggestion list in WIDES (see Section 5.1).

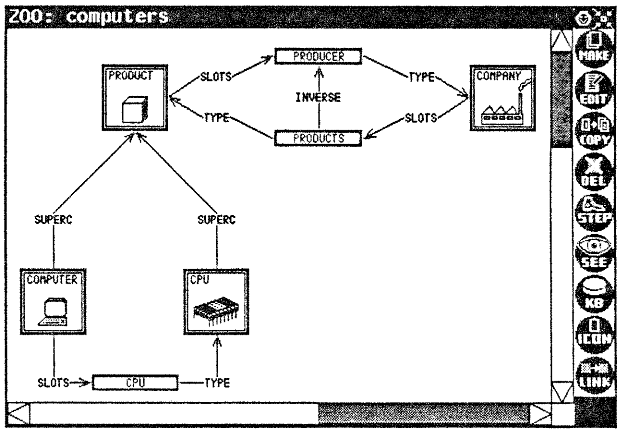


Figure 4-5: The knowledge editor ZOO

5. Design Kits

Powerful construction kits are complex systems containing many different components that can be combined in many ways. In the domain of user interface design, WLISP provides a large number of abstractions. They are a prerequisite, but good user interface components do not guarantee by themselves that they are used at the right place and in the right way. Tools are needed to aid in making design decisions, carry out low-level details, analyze or criticize intermediate versions, and visualize their structure. These tools incorporate knowledge which goes beyond what went into the design of individual components. Specifically, they have additional domain knowledge to aid in the design of *reasonable* artifacts. *Design kits* are steps in this direction.

Currently, the use of WLISP (Section 4.2) requires considerable expertise on the implementation level (i.e., how do I achieve a desired system behavior?) as well as on the domain level (i.e., which user interface technique should be used?). This expertise has to be acquired through an extended learning and experimentation period. To reduce this delay, we have constructed a number of design kits to support the modification and construction of new systems from sets of predefined components. In contrast to simple software construction kits (e.g., the Pinball and Music Construction Kits described previously), which present the designer with the available parts and operations for putting them together and which allow to run the resulting system, design kits give additional support. They incorporate knowledge about which components fit together and how they do so, and they may serve as a critic that recognizes errors or inefficient or useless structures. They are able to deal with multiple representations of the design including drafts, program code, and graphical representations. Design kits constrain the problem space, leaving beginners with fewer choices by providing defaults and grouping the available functions.

Design kits considerably reduce the amount of knowledge a designer has to acquire before useful work can be done. This is especially important if the design environment contains many special purpose components and if each of them is used rarely, even by a full-time designer.

The following two sections describe two design kits for specific areas of the WLISP construction kit. WIDES is a design kit for basic characteristics of window types, and TRIKIT is a design kit for graph display and edit tools.

5.1 WIDES: A Window Design Kit

Because almost all modern user interfaces are window-based, one of the major tasks of user interface design is the definition of a suitable combination of window types. Many current window systems and user interface tool kits offer a wide variety of components such as text, graphic, and network windows and editors, and controls like menus and push buttons. The goals of WIDES are:

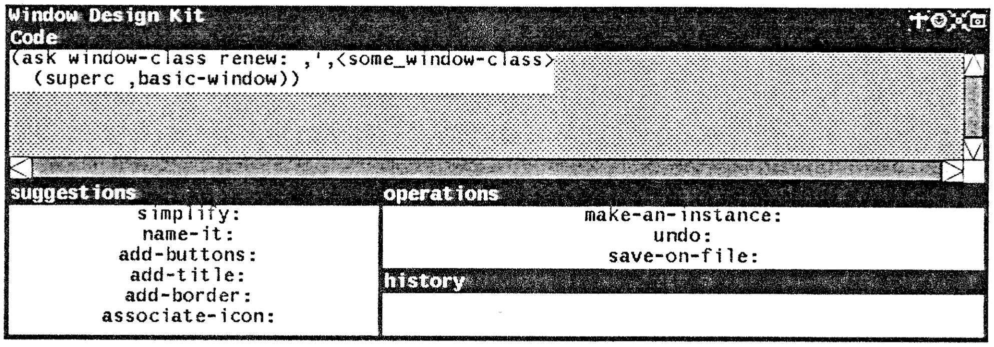
1. to provide a level of abstraction above the object-oriented implementation of these components,
2. to reduce the knowledge required to use the components,
3. to make their use more effective by preventing errors and suggesting the "right" components to use,
4. to support the acquisition of expertise in using these tools.

WIDES provides a safe learning environment in which no fatal errors are possible and in which enough information is provided in each situation to ensure that there is always a way to proceed. The design kit allows its users to create specific window types for their applications.

In the following we will give an example of the use of WIDES and discuss its merits and shortcomings.

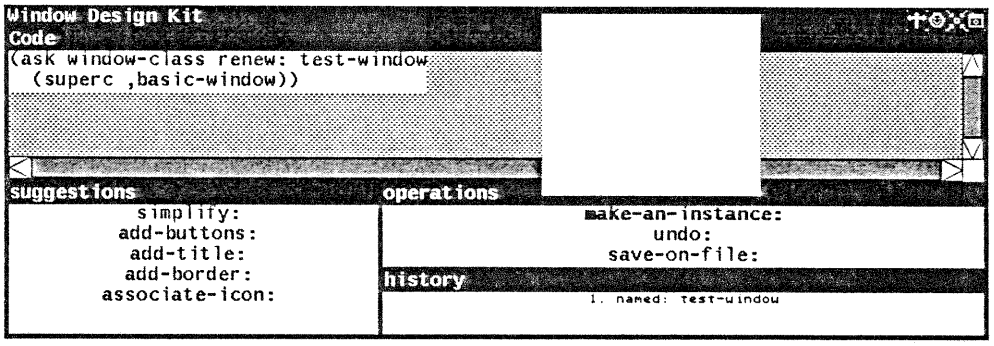
Description of WIDES. The initial state of the system is shown in Figure 5-1. It is a window with four panes:

- a *code* pane that displays the current definition of the window type,
- a menu of *suggestions* for enhancements of the window type,
- a history list, and
- a menu of general *operations*.



```
(ask window-class renew: ,',<some_window-class>
  (superc ,basic-window))
```

Figure 5-1: Initial state of WIDES



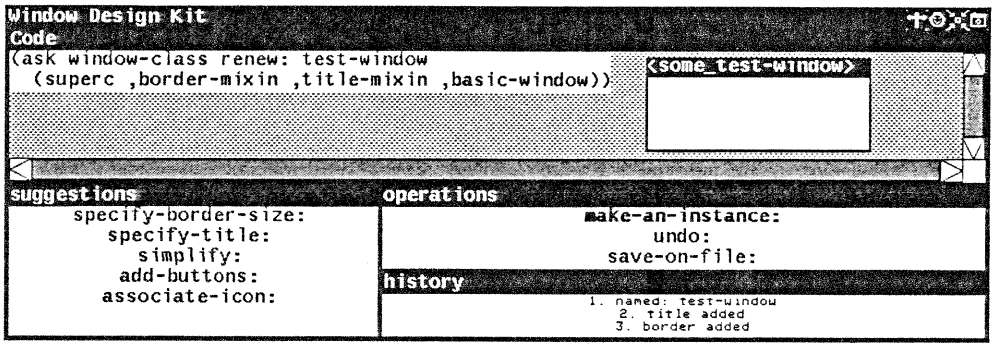
```
(ask window-class renew: test-window
  (superc ,basic-window))
```

```
1. named: test-window
```

Figure 5-2: An instance of the current window definition has been created

Selection of the `name-it:` entry of the suggestions menu makes the system ask for a name for the window type to be built. Selection of the `make-an-instance:` item of the operations menu creates a window that is an *instance* of the type and that corresponds to the current definition in the code pane. This definition describes a very basic type of window (the white rectangle in Figure 5-2); there is no border, no title bar yet; just a rectangular white area. Nevertheless, this window has a set of properties that are inherited from its superclass `basic-window`. It reacts on mouse clicks by showing the `window-menu`, a menu with operations like `move` and `reshape`.

Selection of `add-title:` and `add-border:` produces the state of Figure 5-3. Two superclasses, `border-mixin` and `title-mixin`, have been added to the definition; and a new instance shows a



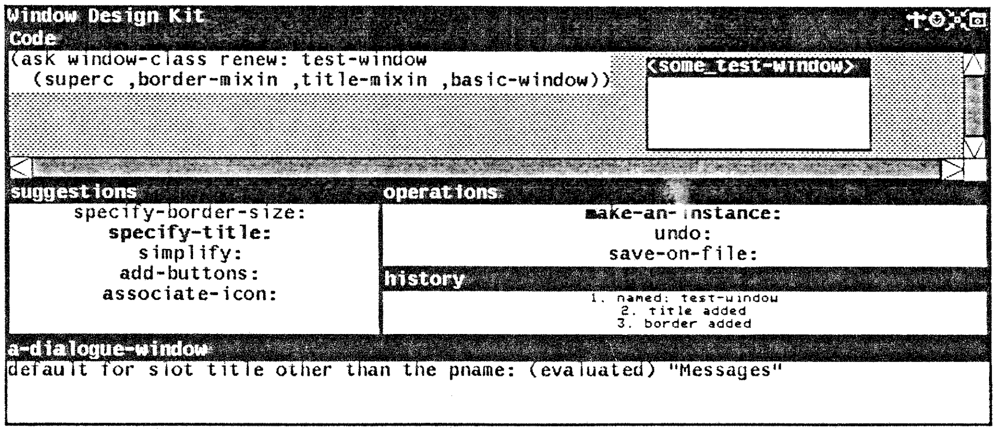
```
(ask window-class renew: test-window
  (superc ,border-mixin ,title-mixin ,basic-window))
```

```
1. named: test-window
2. title added
3. border added
```

Figure 5-3: Title and border have been added to the window type

default title, `some_test-window`, and a default border size of two pixels.

The suggestions menu changes its contents. If, for instance, `add-title:` has been executed, it is replaced by `specify-title`, which would not have been meaningful before having a title. The system, in giving its suggestions, adheres basically to a tree-like regime. Once a key decision like having a title has been made, its menu item is replaced by suggestions for more detailed descriptions. This feature provides the user with some guidance about reasonable next steps, eliminates illegal operations, and reduces the information overload of too many options.



```
(ask window-class renew: test-window
  (superc ,border-mixin ,title-mixin ,basic-window))
```

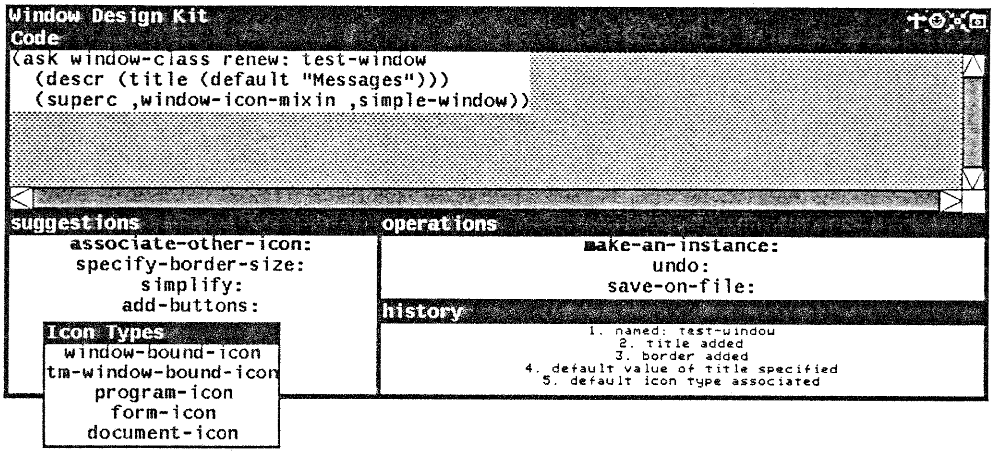
```
1. named: test-window
2. title added
3. border added
```

```
default for slot title other than the pname: (evaluated) "Messages"
```

Figure 5-4: Specification of the title

The next figure (Figure 5-4) shows a modification that requires user input. Selection of `specify-title:` causes a dialogue window to pop up, which prompts the user for an expression to be used as the title of the window. Figure 5-5 shows that the input has been added as a default for the title slot.

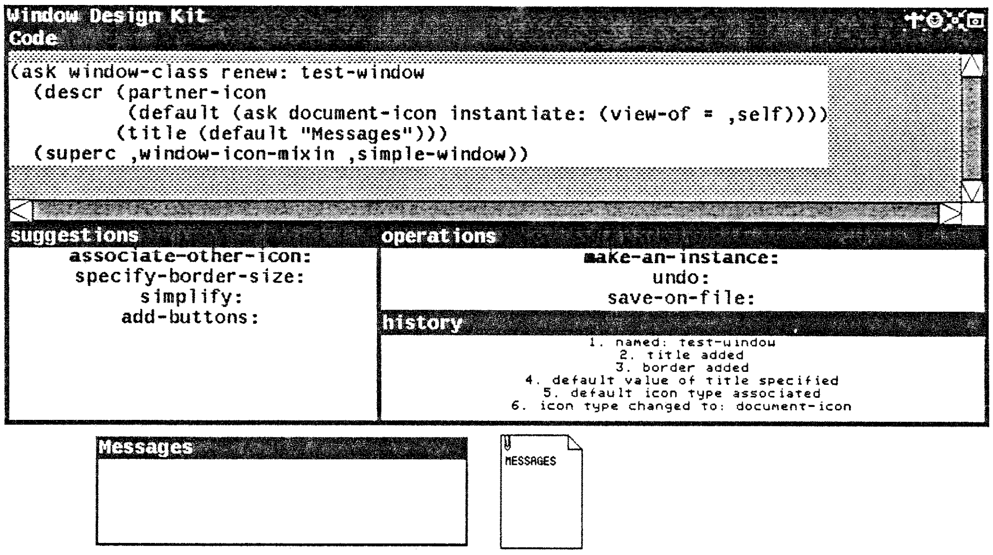
If specific inputs are required, we cannot expect the user to know what the legal inputs are. Therefore, as in Figure 5-5 in which the user associates the window with an icon, a menu of alternatives is displayed (see the pop-up menu at the bottom). Figure 5-6 shows a window and an icon of the selected type (`document-icon`).



```
(ask window-class renew: test-window
  (descr (title (default "Messages")))
  (superc ,window-icon-mixin ,simple-window))
```

```
1. named: test-window
2. title added
3. border added
4. default value of title specified
5. default icon type associated
```

Figure 5-5: Association of a different icon to the window



```
(ask window-class renew: test-window
  (descr (partner-icon
         (default (ask document-icon instantiate: (view-of = ,self))))
        (title (default "Messages")))
  (superc ,window-icon-mixin ,simple-window))
```

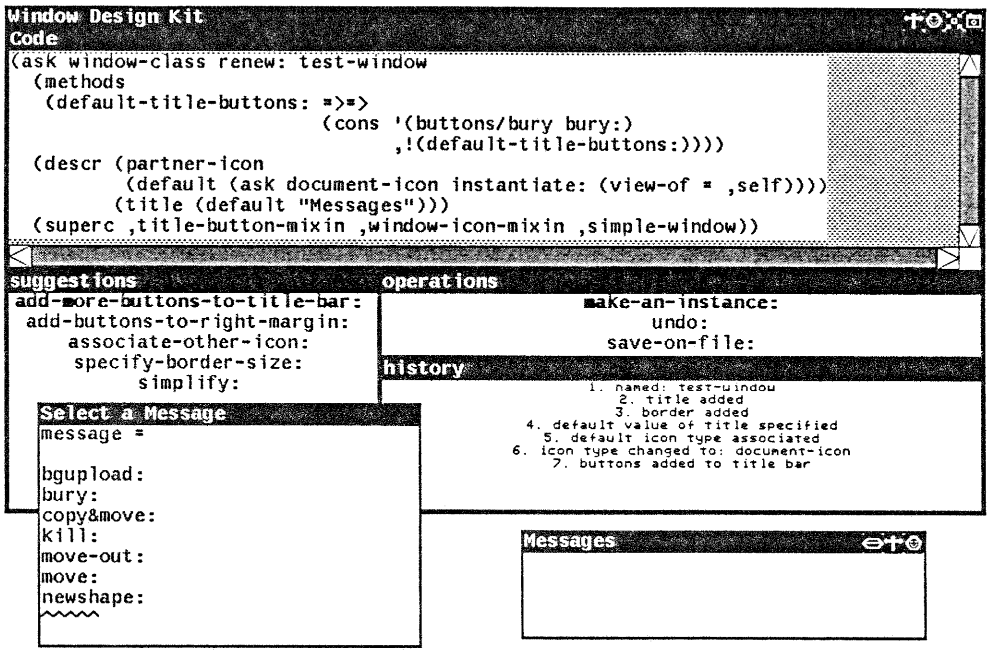
```
1. named: test-window
2. title added
3. border added
4. default value of title specified
5. default icon type associated
6. icon type changed to: document-icon
```

Figure 5-6: The window and its associated icon

An even more complex modification is demonstrated in Figure 5-7. Windows can be associated with buttons such as those in the upper right corner of the WIDES window. Clicking a button with the mouse causes a message to be sent to the window. Selecting `add-buttons:` adds the default set of the two right-most buttons (for `kill` and `refresh`) in the title bar of the messages window. Additional buttons can be defined by selecting `add-more-buttons-to-title-bar:`. This command causes two menus to pop up for selecting a button icon and an associated message. The selection of the message is shown in the figure. The new button in the messages window depicts a coffin for the function of burying or hiding the window. Both menus have, in addition to selecting one of the listed choices, the option of choosing an item not listed in the menu by name.

The `save-on-file:` operation may be used to save the final definition for later use.

Discussion. Although not much code is being generated by the system because it can use many high-



```
(ask window-class renew: test-window
  (methods
   (default-title-buttons: =>=>
                           (cons '(buttons/bury bury:)
                                 ,!(default-title-buttons:))))
  (descr (partner-icon
         (default (ask document-icon instantiate: (view-of = ,self)))
        (title (default "Messages")))
  (superc ,title-button-mixin ,window-icon-mixin ,simple-window))
```

```
1. named: test-window
2. title added
3. border added
4. default value of title specified
5. default icon type associated
6. icon type changed to: document-icon
7. buttons added to title bar
```

Figure 5-7: Adding a button to the title bar

level building blocks (see the code panes in the various stages of the design process), WIDES represents a significant advantage for the user. In order to construct a new window type, it is no longer necessary to know what building blocks (*superclasses*: e.g., `title-mixin`) exist, what their names are, and how they are applied. It is no longer necessary to know that new superclasses have to be added to the `superc` description of a class. Also, WIDES has knowledge about the correct order of the superclasses, what types of icons are available, and the way an icon is associated with a window.

WIDES does not support direct editing of the generated code in the code pane. Analyzing arbitrary program code is generally hard because the code level is more powerful than WIDES. WIDES may not be able to recognize features of the code and accordingly adapt its behavior.

User interface techniques like prompting and menus make it easy to experiment with window construction. The system makes sure that errors are impossible. Although these techniques do not guarantee that users always *understand* what they are doing. In the present system, a test subject could not tell the difference between the `add-title:` and `specify-title:` suggestions before actually trying them. The name `add-title-bar:` probably would have been more appropriate than `add-title:`. Similarly, the pop-up menu of available icon types does not show what the icons look like and it should be replaced by a pictorial menu.

On the other hand, we claim that it is not appropriate to invest too much in making sure that the desired result is achieved with the first try. The system is intended to support an experimental style, and the `undo:` operation should make it easy to step back and retract a decision in order to proceed differently. So far, the UNDO feature has not been implemented, and the question is whether a simple stack-oriented scheme or a selective UNDO of operations further back in the history can be implemented. To support the full use of an UNDO, the system needs a network to take care of dependencies; for example, removing the title bars of a window implies that the buttons have to be removed, too.

The current implementation makes it difficult for a novice to see which modifications of the definition were caused by an action. Highlighting the modifications caused by the last action is a possibility. It should also be possible to point at a piece of the code and obtain an explanation of its function as well as the user action that created it. How can these selections be done? What if the user selects a piece of code that does not correspond clearly to one feature?

Informal experiments with novices have shown that the class - instance abstraction gap is a problem. The code window shows the window class, whereas the windows created by `make-an-instance:` are its instances. A modification of the class (e.g., a modification of a default) does not automatically affect its instances. Properties cannot be specified as immediate values but must be specified as defaults or methods inherited by instances. If the user, experimenting with an instance of the window type, changes the local value of a parameter (e.g., the title) so that it no longer corresponds to the default, then a change of the default in the class has no consequences for existing instances.

The system in its current form is almost too small to be practical. The created window types do not have much functionality and represent only a framework which has to be augmented by more ObjTalk code. Still, users found it exciting that, with some menu selections, real code could be produced. The system also achieves the goal of being a learning tool. It provides a good first impression of the concepts and structure of the domain of constructing window types.

5.2 TRIKIT

A very common user interface problem is the display and modification of hierarchical and network structures. Application systems that deal with rule dependency graphs (Figure 5-19), concepts of a domain of expertise (for explanation purposes), goal trees, or inheritance hierarchies in object-oriented languages are examples in which this problem occurs.

Our response to this problem was to build a design kit for graph display and edit tools. Figure 5-8 illustrates its usage. The application programmer, who is an expert for the application system but not for building user interfaces, sets and adjusts parameters of a generic tool, TRISTAN, and specifies the links between it and the application. The result of the design process is a new, application-specific tool for displaying and editing a network data structure.

The system has been used to build the following applications:

- ObjTalk inheritance hierarchy editor,
- UNIX directory editor (see Figure 5-9),
- Subwindow hierarchy display,
- A project team hierarchy,
- EMYCIN rule dependency display (see Figure 5-19).

Application Domain of TriKit. Many computer programs use data structures that can be viewed as graphs. The nodes are data items that are interconnected by lines representing a semantic relationship between them. In this section we will use the example of a hierarchical file system, which may be displayed as in Figure 5-9. Here the nodes are directories and files which are the leaf nodes. The lines represent the membership relation between files and directories, which can themselves be members of other directories. Each of the nodes is a data structure with properties like name, creation time, owner,

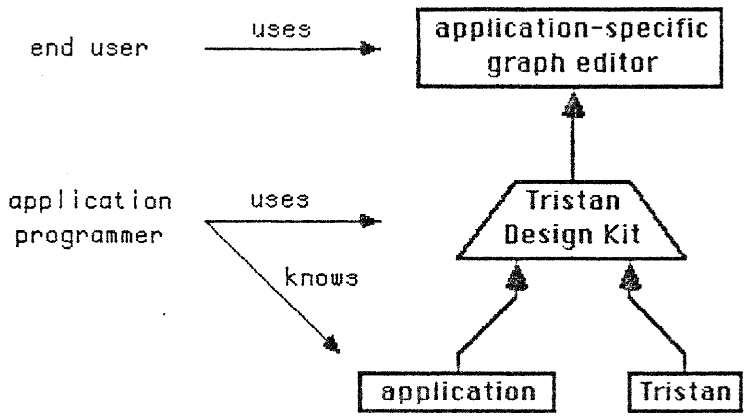


Figure 5-8: Usage of TRIKIT

protection, and size. There are operations to retrieve pieces of the graph (e.g., `list-directory`) and to create and delete nodes and lines.

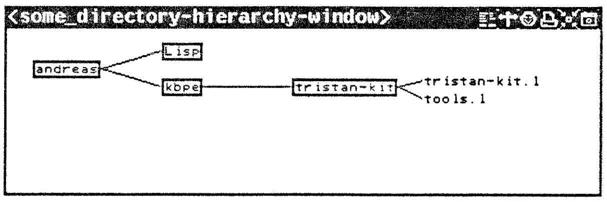


Figure 5-9: A hierarchical file system display

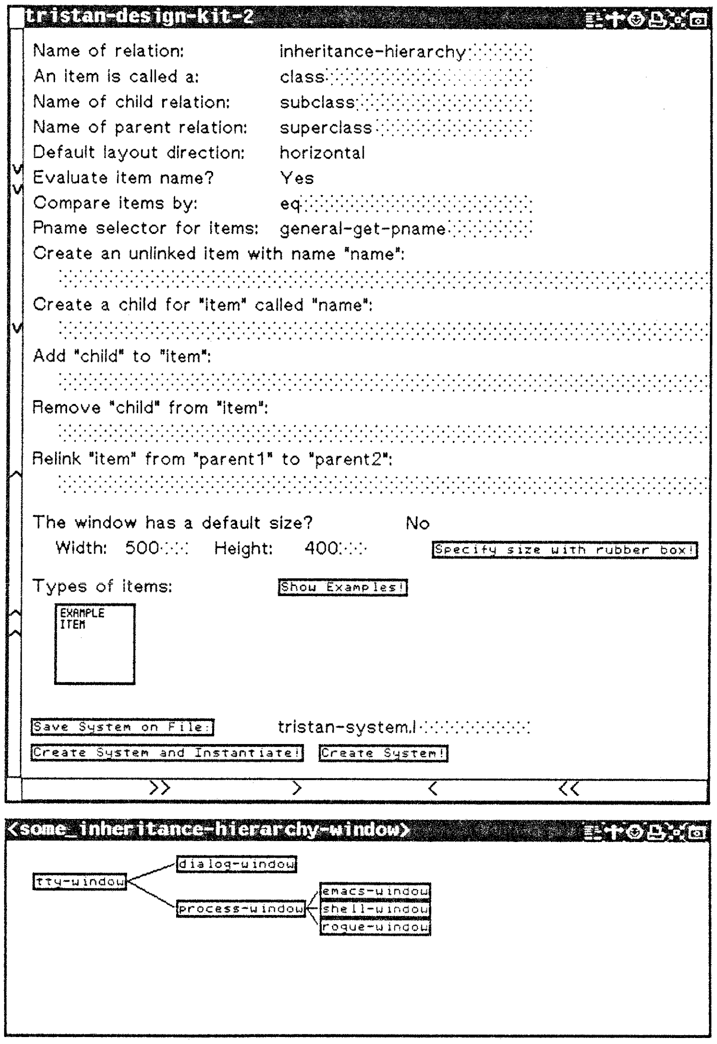
Also, the user must be able to refer to particular nodes of the structure, either by a name relative to some "current node," by an absolute path name specifying the way from a root of the hierarchy, or by some other description. Conversely, a screen representation must be defined for the nodes. This representation might be just the name of the node or the name plus some of the properties, such as owner or size. If there are multiple types of nodes, different representations may be desired (e.g., directories and files in Figure 5-9).

The Generic Graph Display/Editor. TRIKIT is based on TRISTAN [Nieper 85], a facility of WLISP for building direct manipulation display and editing systems for graph structures. TRISTAN supports the following operations:

- selective display of parts of the graph including a node specified by name, immediate children or parents of a node, and a whole subhierarchy of a node (possibly to a certain depth),
- automatic layout planning,
- manual layout modification by constrained moving of nodes,
- highlighting of nodes, and
- editing the graph structure by creating and removing links and nodes.

TRISTAN is independent of the particular node representation. It assumes only that the representation is a subclass of a certain general WLISP window class.

Description of TriKit. TRIKIT presents itself to the user as an interaction sheet as shown in Figure 5-10 (top window). In this window, the user specifies the interface to the application, chooses a graphical representation for the nodes, and controls the creation of the user interface.



tristan-design-kit-2

Name of relation:	inheritance-hierarchy
An item is called a:	class
Name of child relation:	subclass
Name of parent relation:	superclass
Default layout direction:	horizontal
Evaluate item name?	Yes
Compare items by:	eq
Pname selector for items:	general-get-pname

Create an unlinked item with name "name":

Create a child for "item" called "name":

Add "child" to "item":

Remove "child" from "item":

Relink "item" from "parent1" to "parent2":

The window has a default size? No

Width: 500 Height: 400

Types of items:

tristan-system.l

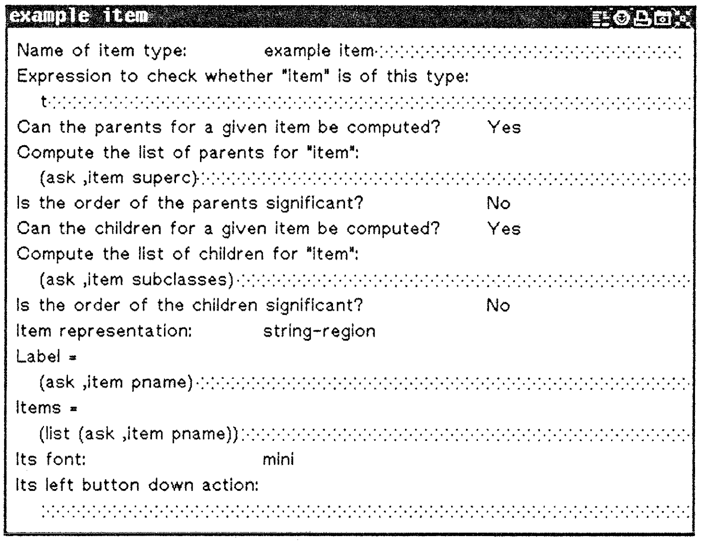
Figure 5-10: Initial state of the main form and an inheritance hierarchy window generated from it

The following types of fields may be found in the interaction sheet:

`edit fields`	indicated by their dotted background; for entry and modification of names, numbers, program code, etc.; a mouse click on the field moves the cursor into it and allows editing of its contents.
`choice fields`	if the number of possible values of a field is very small, this type of field is being used; mouse clicks circle through the set of values.
`menu fields`	for a larger number of choices; a mouse click produces a pop up menu.
`push buttons`	low and long rectangles with a black frame; a mouse click activates their associated action.

`subform icons` large squares; a mouse click produces a subform.

The initial form is filled in with an example application of an ObjTalk inheritance hierarchy display system (Figure 5-10, bottom window). This allows users to familiarize themselves with TRIKIT, to modify parameters, and to learn their significance.



```
example item
Name of item type:          example item
Expression to check whether "item" is of this type:
   t
Can the parents for a given item be computed?     Yes
Compute the list of parents for "item":
   (ask ,item superc)
Is the order of the parents significant?          No
Can the children for a given item be computed?    Yes
Compute the list of children for "item":
   (ask ,item subclasses)
Is the order of the children significant?         No
Item representation:        string-region
Label =
   (ask ,item pname)
Items =
   (list (ask ,item pname))
Its font:                   mini
Its left button down action:
```

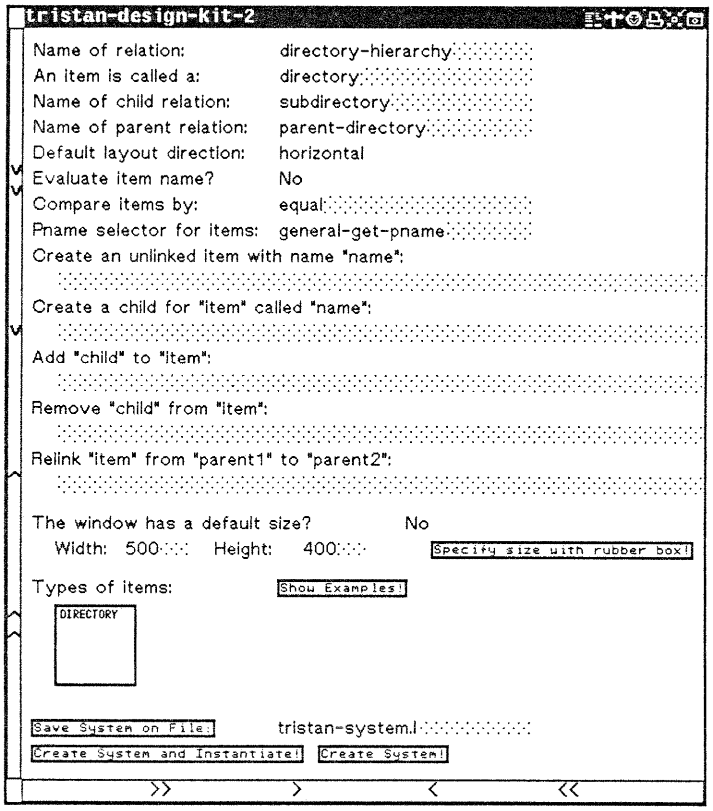
Figure 5-11: Initial state of the node form

Clicking the square representing the `example item` subform produces the form of Figure 5-11. While the main form is associated with the graph in general, the subforms describe the properties of its nodes.

Let us examine the use of the system through the example of building a directory editor like the one shown in Figure 5-9. A directory editor is a tool for viewing a hierarchical file system and for doing operations on it such as creating or removing a directory, moving a file into another directory, and renaming files.

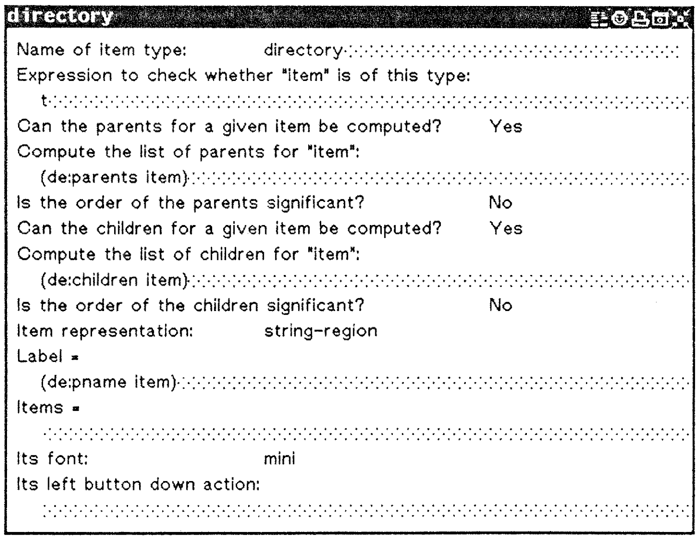
In Figure 5-12 the first four fields have been filled in to reflect the terms of the file system domain. They establish a common vocabulary for the user and the system. They describe the names of the relation to be displayed, the names of the items that are elements of the relation, and those of the links to superordinate and subordinate nodes in the relation. The next field, `Evaluate item name?`, says that a user-entered name of a file or directory represents itself as opposed to being the name of a variable holding the actual item. `Equal` is used as a comparison function for directory names. No other changes have been made to this form.

The `example item` form has been renamed (Figure 5-13), and the two most important fields have been adapted to the new application: The `de:parents` function computes the list of superdirectories and the `de:children` function computes the subdirectories, that is, the contents of the directory. The `de:pname` function in the label field computes a "print name", or a label, for the items; that is, it strips off the leading pathname component and leaves the file name, which is unique only locally within its directory.



Name of relation:	directory-hierarchy
An item is called a:	directory
Name of child relation:	subdirectory
Name of parent relation:	parent-directory
Default layout direction:	horizontal
Evaluate item name?	No
Compare items by:	equal
Pname selector for items:	general-get-pname
Create an unlinked item with name "name":	
Create a child for "item" called "name":	
Add "child" to "item":	
Remove "child" from "item":	
Relink "item" from "parent1" to "parent2":	
The window has a default size?	No
Width:	500
Height:	400
Types of items:	DIRECTORY
Save System on File:	tristan-system.l

Figure 5-12: Main form, modified to describe a directory editor



Name of item type:	directory
Expression to check whether "item" is of this type:	t
Can the parents for a given item be computed?	Yes
Compute the list of parents for "item":	(de:parents item)
Is the order of the parents significant?	No
Can the children for a given item be computed?	Yes
Compute the list of children for "item":	(de:children item)
Is the order of the children significant?	No
Item representation:	string-region
Label =	(de:pname item)
Items =	
Its font:	mini
Its left button down action:	

Figure 5-13: Node form, modified to describe a directory node

Functions with a "de:" prefix (de:parents, de:children, de:pname) belong to the application domain. They are application-specific and must be supplied by the application programmer.

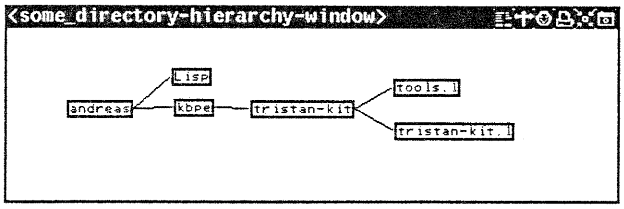


Figure 5-14: An example directory hierarchy window

The modifications mentioned previously are sufficient to produce an initial working version of the directory editor. A click on the "`Create System and Instantiate!`" push button compiles the forms into a TRISTAN system. Part of it is the directory-hierarchy window type, which is being instantiated to produce the window of Figure 5-14. It shows a directory called `andreas` with two of its subdirectories, `Lisp` and `kbpe`. `kbpe` has further subdirectories. Note that it is not necessary to display the complete graph; the display may be limited to any subset of the whole.

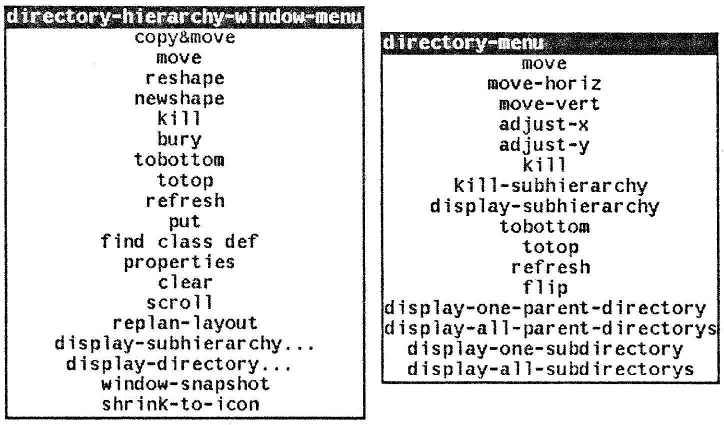


Figure 5-15: The main menus of the directory hierarchy window and the directory node

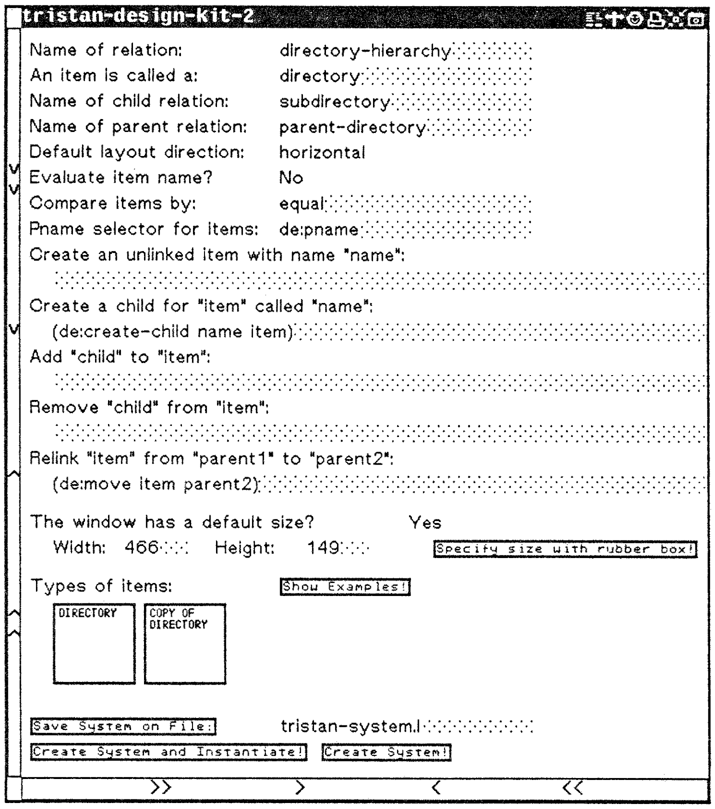
Figure 5-15 shows on the left the main menu of directory hierarchy windows. In addition to generic window operations (top items up to `scroll` and the two bottom items), there are three application specific operations that have been added automatically:

`replan-layout`
: supplied by the TRISTAN system: automatically rearranges the layout of the graph;

`display-subhierarchy...`
: also supplied by the TRISTAN system: displays all the recursively subordinate nodes of a given node;

`display-directory...`
: generated by TRIKIT: displays a given directory. This is a renamed version of the generic `display-item...` operation of TRISTAN.

The menu on the right of Figure 5-15 is associated with individual directory nodes. Again, some of the operations, like `move`, are general. Others, like `kill-subhierarchy`, are supplied by TRISTAN, and the display operations are generated by TRIKIT.



tristan-design-kit-2

Name of relation:	directory-hierarchy
An item is called a:	directory
Name of child relation:	subdirectory
Name of parent relation:	parent-directory
Default layout direction:	horizontal
Evaluate item name?	No
Compare items by:	equal
Pname selector for items:	de:pname

Create an unlinked item with name "name":

Create a child for "item" called "name":
(de:create-child name item)

Add "child" to "item":

Remove "child" from "item":

Relink "item" from "parent1" to "parent2":
(de:move item parent2)

The window has a default size? Yes
Width: 466 Height: 149

Types of items:

tristan-system.l

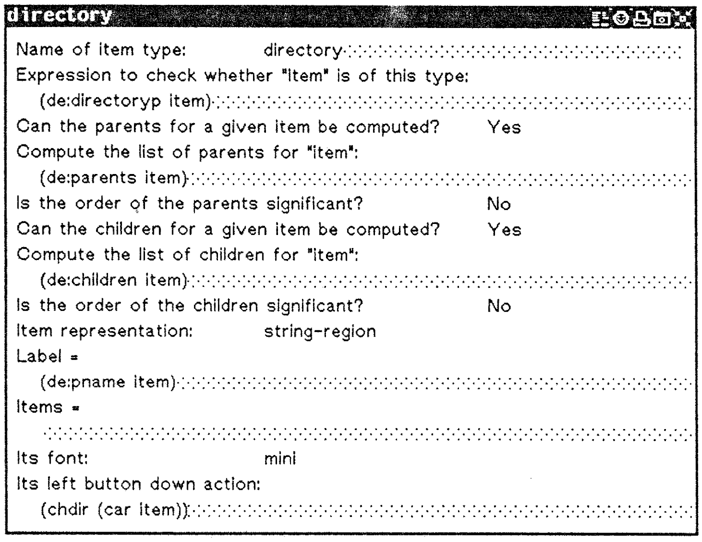
Figure 5-16: Extended description of the directory hierarchy window

Currently the directory hierarchy editor has a number of shortcomings. If the system is to be a true editor, it should be possible to create new nodes and to alter the graph structure. For this purpose, the meaning of creating a child and of relinking a node from one parent to another have been specified in the main form using the application functions `de:create` and `de:move` (Figure 5-16).

Also, there are actually two types of nodes in the application: directories and files. The user therefore creates a new subform (`Copy of Directory`) by cloning the existing one. Now the system must be told how to distinguish between the two node types. Consider Figure 5-17, in which the second field specifies the necessary predicate expression (`de:directoryp`). There has also been an action associated with the node: clicking the node will make the directory it is representing the current working directory.

This action is not appropriate for plain files (Figure 5-18). Instead, an action to load the file into an editor has been specified. Because the nodes are semantically different, it would be nice to display them differently. The `Item representation` field has been set to `label-region`, which has no frame. A window created according to these modifications now looks exactly like Figure 5-9 on page 21.

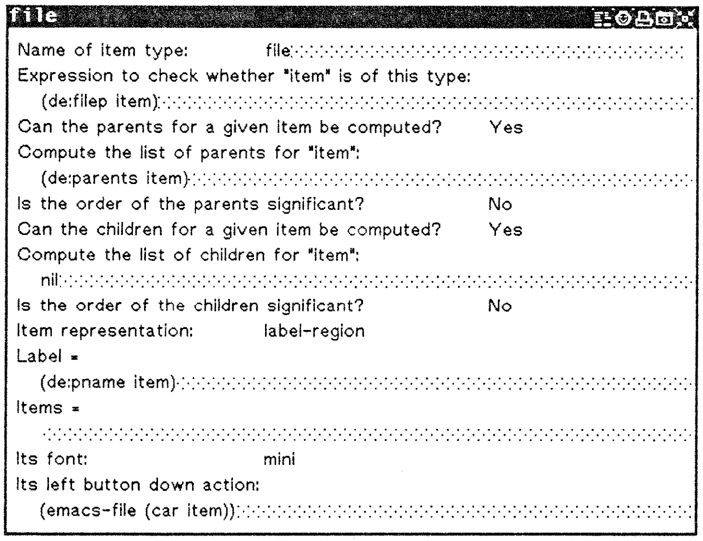
If this version of the system is not yet satisfactory, the user will have to work on the system at the implementation level of ObjTalk and LISP. The `Save System on File` operation in the main form creates a file containing the code of the directory editor, which may then be further modified.


directory

Name of item type: directory
Expression to check whether "item" is of this type:
(de:directoryp item)
Can the parents for a given item be computed? Yes
Compute the list of parents for "item":
(de:parents item)
Is the order of the parents significant? No
Can the children for a given item be computed? Yes
Compute the list of children for "item":
(de:children item)
Is the order of the children significant? No
Item representation: string-region
Label =
(de:pname item)
Items =

Its font: mini
Its left button down action:
(chdir (car item))

Figure 5-17: The directory node form


file

Name of item type: file
Expression to check whether "item" is of this type:
(de:filep item)
Can the parents for a given item be computed? Yes
Compute the list of parents for "item":
(de:parents item)
Is the order of the parents significant? No
Can the children for a given item be computed? Yes
Compute the list of children for "item":
nil
Is the order of the children significant? No
Item representation: label-region
Label =
(de:pname item)
Items =

Its font: mini
Its left button down action:
(emacs-file (car item))

Figure 5-18: The plain file node form

A Rule Dependency Display: Experiences with TriKit. This section reports our experiences with the application of TRIKIT to the visualization of the relationship of rules of in an EMYCIN-like formalism. One of our students used TRIKIT to build the rule dependency display for scanning and debugging a knowledge base of approximately 100 rules (Figure 5-19). A rule is displayed as a small rectangle containing the rule number and, on its left, the premises and on the right the conclusions. Both of them are displayed as larger rectangles with several lines of text.

Separate rules can share nodes within this network. For example, a conclusion of one rule can serve as a premise of another rule.

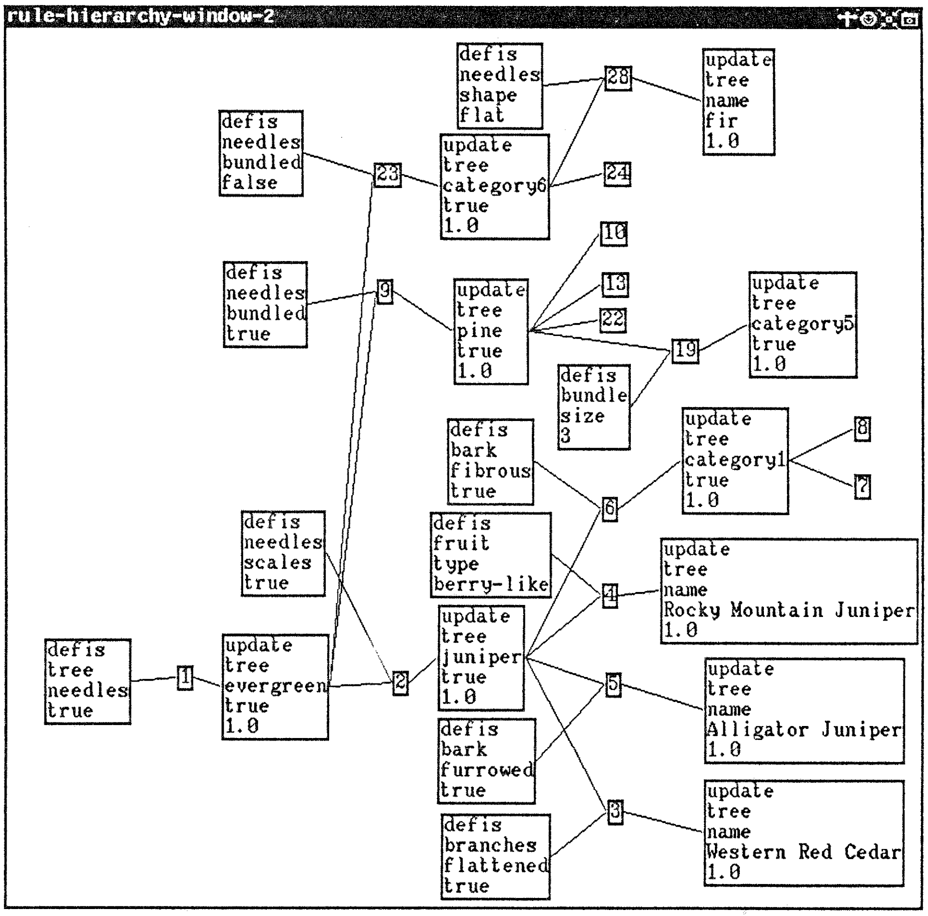

rule-hierarchy-window-2
defis needles shape flat
update tree name fir 1.0
defis needles bundled false
23
update tree category6 true 1.0
28
24
defis needles bundled true
9
update tree pine true 1.0
10
13
22
19
update tree category5 true 1.0
defis bundle size 3
defis bark fibrous true
update tree category1 true 1.0
8
7
6
defis needles scales true
defis fruit type berry-like
4
update tree name Rocky Mountain Juniper 1.0
defis tree needles true
1
update tree evergreen true 1.0
2
update tree juniper true 1.0
5
update tree name Alligator Juniper 1.0
defis bark furrowed true
defis branches flattened true
3
update tree name Western Red Cedar 1.0

Figure 5-19: A rule dependency display

Using the system uncovered problems with certain features and concepts that were not initially understood:

- The system can be created, instantiated and saved. The users need to understand what is being created and what is being saved (e.g., a definition of a hierarchy? the forms? their application? a specific picture?).
- It is not obvious at first that to define a relation two types of sheets are needed, one relating to the hierarchy in general and the second (displayed by clicking `example node`) defining the different nodes in the system. The names of the sheets should be changed to better reflect their purpose.
- The user may not realize that the system contains default values in the main hierarchy sheet.
- In the sheets the two terms *item* and *name* are used. It is not clear that *item* refers to the elements of the application relation as data structures, whereas *name* refers to their external representation through an identifier.
- Some parts of the system can be understood only through experimentation. An example is the `item representation` field of the node subform, whose possible values (e.g., `string-region`, `label-field`) determine the overall appearance of the nodes.
- The meaning of some fields, such as `Pname selector for items`, is not obvious. Also, the user may not know whether a system-supplied value of a field (`general-get-pname`)

works with his or her application.

- The use of the fields is not obvious. Some values of fields have functional purposes and some serve just as labels, for example, as the title of a menu.

Because of these problems and because no additional documentation was available, the student needed some help from the system designer. Once he understood those issues, however, he was able to extend the rule display independently and make good use of it. This experiment gave us valuable insights into the weak points of TRIKIT.

Discussion. With TRIKIT the user can construct useful systems without knowing the details of the selected building blocks. This design process happens on the level of abstract properties of graphs (e.g., horizontal versus vertical layout of the graph), not on the implementation level (ObjTalk classes). Sometimes it is necessary to use code-level specifications, for instance, code that computes the list of parents for an item (Figure 5-13). This form of specification does not pertain to the graph as such, but is required for the interface between application and TRIKIT; TRIKIT does not make any assumption about how graphs might be implemented in application systems.

A critical design decision was whether to make the user explicitly aware of the distinction between an item in his or her relation and the node object representing it as a labeled rectangle in the display. Initially we made this distinction. But later we dropped it from the model of the design process presented to the user. This is an example of the abstraction from implementation details that we hope to achieve with design kits like TRIKIT.

Although the design space of TRIKIT is limited by the available options in the forms, it is possible to use this system to create a prototype, which may be refined on a lower level. The design space of possible systems could be extended in several ways. One way of doing this is extending the forms with new fields and providing new forms for other aspects or views of the system. The tool could be expanded to:

- visualize dynamic processes by highlighting currently involved nodes (inferences in the rule display),
- allow output to be sent to nodes,
- support more than one type of link between nodes,
- be integrated with efforts to build a more general window design kit (Section 5.1). This capability could give the application programmer more control over the behavior of the graph window and the individual nodes. The generated graph display/editor could be made a part of a larger system.

Further research will also be done to incorporate general knowledge of graph data structures. Given an example of a graph, or a description of the data structure such as a type definition or a MYCIN context definition, the system could automatically produce an initial display design.

5.3 Comparison

One major goal in the development of the TRIKIT system was to overcome the limitation of the WIDES suggestions menu as being the only input mode. The command interaction style provided by the suggestions menu of WIDES is replaced in TRIKIT with a declarative or descriptive mode.

In WIDES it should be possible to disregard the suggestions and explore other parts of the design that the system does not currently consider important. This feature should be an additional alternative for the expert, because the suggestions menu makes interaction with WIDES straightforward for novices. TRIKIT, on the other hand, lacks this kind of guidance, and users are in danger of being overwhelmed with the many options, not knowing which of them will be important for getting done a first version of an application.

With WIDES the users are aware that code is being generated and they can learn the functions of the different pieces of code. This makes it easy to make enhancements on the implementation level. Users are incrementally writing code (creating a definition) by selecting high-level commands from a dynamic menu. TRIKIT does not show the code level to the user at all. This is because TRIKIT is itself powerful enough to generate complete, working systems. Code is generated internally from the specification represented by the filled in forms.

Both systems contain knowledge about their respective domains. They provide a subgoal decomposition that identifies subproblems of the design space in which the user wants to solve problems (Figure 5-20). The design kits use the partial solutions to build up a complete design. In order to achieve that components work together properly, they must be ordered correctly (e.g., the `title-mixin` must always precede the `border-mixin`). Existing objects must be notified, when new objects are created (e.g., the `create-child` operation notifies its superordinate node). This knowledge is represented in the procedures of the design kits that generate the code. More explicit knowledge representations are required to make the design kits themselves extensible.

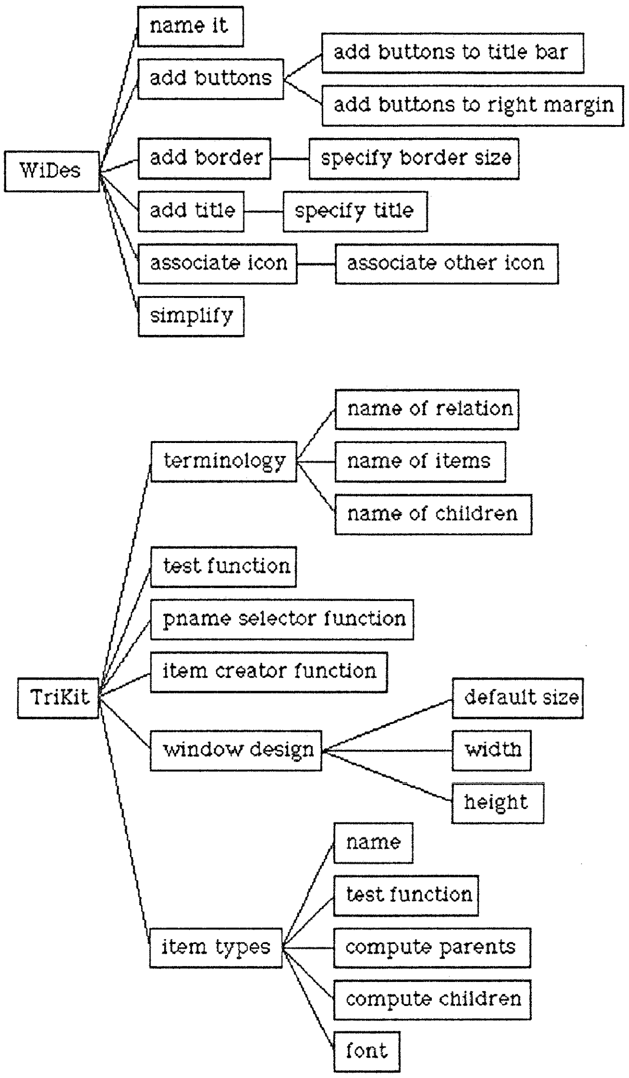


Figure 5-20: Subgoal trees for WIDES and TRIKIT

6. Assessment of our Efforts

The human-computer interaction subsystem is crucial to the usability and success of most computer systems. New prescriptive goals (e.g., convivial and symbiotic systems), new methodologies (e.g., differential programming, use of kits) and new tools (e.g., intelligent support systems) are needed to make systems usable and useful.

Some of the important tradeoffs and design problems in this area of research are:

- the usability barrier: it takes a major effort and large training costs to learn a complex system that offers extensive functionality;
- the construction kit versus complete system decision: whether to design the system completely in advance or to offer a metasystem enabling end-users to alter it according to their needs;
- distribution of control: does the system or the user make decisions? Can control shift back and forth between the agents involved so that cooperative problem-solving is possible?

The advantages of construction and design kits are:

- they provide a powerful environment for rapid prototyping of a large class of systems;
- they increase the control of the user over systems without requiring the user to learn many details;
- the large class of existing building blocks "guarantees" to some extent the construction of high-quality systems with relatively low construction costs; systems can be created more quickly because the designer can rely on well-developed parts and take advantage of stable subassemblies (e.g., exploiting the rich inheritance network in the WLISP system).
- the associated support tools make it easier to learn and work with complex systems.

Specific construction and design kits are used by different classes of users for a variety of different tasks. Our experience has shown that the use of design kits is not restricted to the inexperienced user: if the functionality offered by the design kit is sufficient, then there is no reason why the expert should not use it.

It is misleading to assume that knowing how to use a construction and design kit will come for free and will not require any learning process at all. Learning processes are required at different levels in using the kits: users have to operate on different descriptive levels, they need to understand the domain concepts used in the kits, and they must know how to use a specific kit for their purposes and goals.

It remains an open question whether we can succeed in considerably extending the functionality of the design kits to cover a substantial part of their domains while retaining or even improving the simplicity of use. WIDES is currently easy to use, but we do not know whether we will be able to retain this property when the system covers not only simple kinds of windows but also other objects like menus, icons, gauges.

It has been a difficult problem to establish a shared vocabulary between the designer and the user that enables the user to understand the descriptions for the required inputs. The meaning of the label `Pname selector for items` (Figure 5-16) is not obvious to someone who does not know this technical term of the menu system. To remedy shortcomings like this, we have to solve the following problems:

- find a better conceptualization of the design task and use it to restructure the forms to make them easier to understand;

- select prototypical examples more carefully and convey a better feeling of what needs to be done by taking task structures and the user's knowledge into account;
- provide optionally displays of help texts for all fields;
- allow alternate modes of specification; explore more direct forms of manipulation of prototypes;
- prompt for information at the time it is needed. Information about what it means to create a link between two nodes should be asked for only when this action is being executed for the first time.

Currently the interaction with the design kits is mostly a user's *monologue*. The system does not act on its own initiative on user inputs. Our goal is to make *dialogues* possible in which the user and the system take actions in turn and correct each other's errors and false assumptions. Small examples of this kind of interaction can be found in the current implementations:

- default values of fields represent the design kit's initial assumptions of useful values;
- when the user creates a new type of node in TRIKIT, the system copies its properties from an existing node because the new node is probably more like the existing node than like the original defaults.

Consistency becomes a problem when systems can be modified on different descriptive levels. If design using a kit can be mapped into simple operations such as setting parameters of predefined building blocks on the code level, consistency can easily be achieved (e.g., specifying the size of a window on the code level by giving two constants for width and height). If, however, a part of a definition (e.g., the list of superclasses) is derived from multiple specifications of the user (e.g., whether a title bar or an icon is desired) the design kit cannot, in general, decide how to reconcile the interactive specifications and the code-level specifications. A program analysis component could make it possible for the high-level (form) description and the program code to coexist and for the user to use both languages alternately (for a further discussion of this problem see [Waters 86]).

Other issues arising from the conceptual distance between a description and what it describes need to be further explored. What happens to an existing object when its description has been changed? Should it be updated? Updating is not always desirable or possible because immediate updates may be computationally too expensive, the screen display may change to such a degree that the user loses track of where things are, or changes of descriptions of actions executed at the time an object is created have no effect on already existing objects (e.g., initial size of a window).

Regarding kits as desirable components of computer systems, the question arises what we have to do to develop *kit-kits*, that is, kits for designers of construction and design kits. What challenges and problems do designers face in developing new kits or extending existing ones? Their design requires a qualitatively different descriptive level using many more abstractions and they have to defer commitment so that users are left with their share of and influence on the design.

7. Conclusions

The challenge of increasing the effectiveness of human use of computers goes beyond technical questions. Software must be designed to optimize not the way machines work, but the way people think -- an objective which we hope to meet in part with human problem-domain communication. Just as television, the telescope, and the microscope have amplified our sight, computers will amplify human capabilities by expanding memory, augmenting reasoning, and facilitating communication.

We are interested in building evolutionary systems that grow to fit an environment of needs rather than carrying out a single, well-specified task. In these systems the main activity of programming has moved from the origination of new programs to the modification of existing ones. If designers are to modify existing programs, they must understand how the parts of these programs function. An important question concerns the level of understanding necessary for successful reuse and redesign: exactly how much does the user have to understand? A construction kit with a large number of generally useful building blocks provides a good basis for reuse and redesign; but without the additional assistance of design kits, users are lost in the wealth of information and possibilities offered by complex systems. We believe that human problem-domain communication is the most promising way of overcoming these problems.

Acknowledgments

The authors would like to thank their former colleagues from the INFORM project at the University of Stuttgart: Franz Fabian who has created the foundations of WLISP, and Wolf-Fritz Riekert who developed ZOO, as well as their current colleagues at the University of Colorado, Boulder: Bart Burns who implemented the WFORMS kit, Christopher Morel who applied TRIKIT to the rule dependency display, Helga Nieper who developed TRISTAN, and Christian Rathke who developed FINANZ. Without their contributions, the described research effort would not have been possible.

The research was supported by: grant No. DCR-8420944 from the National Science Foundation, grant No. N00014-85-K-0842 from the Office of Naval Research, and grant No. MDA903-86-C0143 from the Army Research Institute.

References

[Bates, Bobrow 84]
M. Bates, R.J. Bobrow, *Natural Language Interfaces: What's Here, What's Coming, and Who Needs It*, in W. Reitman (ed.), *Artificial Intelligence Applications for Business*, Ablex Publishing Corporation, Norwood, NJ, 1984, pp. 179-194, ch. 10.

[Boecker, Fabian, Lemke 85]
H.-D. Boecker, F. Fabian Jr., A.C. Lemke, *WLisp: A Window Based Programming Environment for FranzLisp*, Proceedings of the First Pan Pacific Computer Conference, The Australian Computer Society, Melbourne, Australia, September 1985, pp. 580-595.

[Boecker, Fischer, Nieper 86]
H.-D. Boecker, G. Fischer, H. Nieper, *The Enhancement of Understanding through Visual Representations*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 44-50.

[Bolt 84] R.A. Bolt, *The Human Interface*, Lifetime Learning Publications, Belmont, CA, 1984.

[Buchanan, Shortliffe 84]
B.G. Buchanan, E.H. Shortliffe, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984.

[Chambers, Nagel 85]
A.B. Chambers, D.C. Nagel, *Pilots of the Future: Human or Computer?*, Communications of the ACM, Vol. 28, No. 11, November 1985.

[Fabian 86]
F. Fabian, *Fenster- und Menuesysteme in der MCK*, in G. Fischer, R. Gunzenhaeuser (eds.), *Methoden und Werkzeuge zur Gestaltung benutzergerechter Computersysteme*, Walter de Gruyter, Berlin - New York, Mensch-Computer-Kommunikation Vol. 1, 1986, pp. 101-119, ch. V.

[Fischer 83a]
G. Fischer, *Symbiotic, Knowledge-Based Computer Support Systems*, Automatica, Vol. 19, No. 6, November 1983, pp. 627-637.

[Fischer 83b]
G. Fischer, *Navigationswerkzeuge in wissensbasierten Systemen*, Office Management, Vol. 31, No. Sonderheft, 1983, pp. 49-52.

[Fischer 84]
G. Fischer, *Human-Computer Communication and Knowledge-Based Systems*, in H.J. Otway, M. Peltu (eds.) *The Managerial Challenge of New Office Technology*, Butterworths, London, 1984, pp. 54-79, ch. 3.

[Fischer 87]
G. Fischer, *A Critic for LISP*, Technical Report, Department of Computer Science, University of Colorado, Boulder, 1987.

[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), North-Holland, 1983, pp. 73-88.

[Fischer, Kintsch 86]
G. Fischer, W. Kintsch, *Theories, Methods and Tools for the Design of User-Centered Systems*, Technical Report, Department of Computer Science, University of Colorado, Boulder, 1986.

[Fischer, Lemke 87]
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, Inc., Hillsdale, NJ, 1987.

[Fischer, Lemke, Rathke 87]
G. Fischer, A.C. Lemke, C. Rathke, *From Design to Redesign*, Proceedings of the 9th International Conference on Software Engineering, IEEE, March 1987.

[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, Schneider 84]
G. Fischer, M. Schneider, *Knowledge-Based Communication Processes in Software Engineering*, Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, March 1984, pp. 358-368.

[Hutchins, Hollan, Norman 86]
E.L. Hutchins, J.D. Hollan, D.A. Norman, *Direct Manipulation Interfaces*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1986, pp. 87-124, ch. 5.

[Nieper 85]
H. Nieper, *TRISTAN: A Generic Display and Editing System for Hierarchical Structures*, Technical Report, Department of Computer Science, University of Colorado, Boulder, 1985.

[Norman, Draper 86]
D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1986.

[Olsen et al. 84]
D.R. Olsen Jr., W. Buxton, R. Ehrich, D.J. Kasik, J.R. Rhyne, J. Sibert, *A Context for User Interface Management*, IEEE Computer Graphics and Applications, December 1984, pp. 33-42.

[Rathke 86]
C. Rathke, *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache*, PhD Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

[Riekert 86]
W.-F. Riekert, *Werkzeuge und Systeme zur Unterstuetzung des Erwerbs und der objektorientierten Modellierung von Wissen*, PhD Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

[Robertson, McCracken, Newell 81]
G. Robertson, D. McCracken, A. Newell, *The ZOG Approach to Man-Machine Communication*, International Journal of Man-Machine Studies, Vol. 14, August 1981, pp. 461-488.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.

[Waters 86]
R.C. Waters, *KBEmacs: Where's the AI?*, AI Magazine, Vol. 7, No. 1, Spring 1986, pp. 47-56.

[Wilensky et al. 84]
R. Wilensky, Y. Arens, D. Chin, *Talking to UNIX in English: An Overview of UC*, Communications of the ACM, Vol. 27, No. 6, June 1984, pp. 574-593.

[Williams et al. 82]
M.D. Williams, F.N. Tou, R. Fikes, A. Henderson, T. Malone, *RABBIT: Cognitive Science in Interface Design*, Proceedings of the Cognitive Science Conference, Ann Arbor, Michigan, 1982.

[Winograd 79]
T. Winograd, *Beyond Programming Languages*, Communications of the ACM, Vol. 22, No. 7, July 1979, pp. 391-401.

[Winograd, Flores 86]
T. Winograd, F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ, 1986.

[Woods 86]
David D. Woods, *Cognitive Technologies: The Design of Joint Human-Machine Cognitive Systems*, AI Magazine, Vol. 6, No. 4, Winter 1986, pp. 86-92.