HUMAN-COMPUTER INTERACTION, 1987-1988, Volume 3, pp. 179-222

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

Gerhard Fischer and **Andreas C. Lemke**
University of Colorado

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. 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 kits and design environments are tools that represent steps toward 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 environments 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.

Authors' present address: Gerhard Fischer and Andreas C. Lemke, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder, CO 80309-0430.

CONTENTS

1. INTRODUCTION

Even for the expert—let alone the novice and occasional user—it is difficult to take advantage of the available computational power to use the computer for a purpose chosen by himself or 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. 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. 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.

Users use computers as tools for achieving tasks of particular problem domains such as text processing, financial planning, or computer-aided design. The fact that in order to communicate with most computer systems,

the user has to learn a new language together with a whole new world of concepts is a reason for the low usability of many powerful systems. These systems restrict users who cannot expend the effort to familiarize themselves with these large syntactic and conceptual worlds to make only marginal use of the systems. Users, instead of being able to structure designs themselves, must make do with what they can produce based on their limited mental model of the system or must delegate work to experts and thereby relinquish their control. These users can be end users with little knowledge about computers as well as programmers wanting to use facilities that are outside the domain of their main interest.

If users can communicate with a computer tool in the language of the problem domain, a language that they were trained to use, then the communication barrier is much lower. We refer to this type of capability as *human problem-domain communication. Convivial tools* allow users to express themselves according to their ideas and to convert these ideas into actions and artifacts (Illich, 1973).

Increasingly, users are working in domains that they were not trained in and that they knew a priori very little about. An example is typesetting and desktop publishing using modern page layout systems. Human problem-domain communication obviously cannot free users from acquiring necessary domain concepts, but it can support communication at an adequate level.

In order to support communication at the domain level, concepts of the problem domain must be represented in the system. We consider two types of support systems—construction kits and design environments—as examples for human problem-domain communication. *Construction kits* make abstractions of the problem domain directly available to the user. They can be combined to achieve the user's task. Construction kits provide syntactic support for this combination process and for the visualization of the results.

A construction kit with a large number of generally useful building blocks provides a good basis for reuse and redesign; but there are two important deficiencies of construction kits: They do not help users understand the components that they provide, and they do not support the application of the components to problem solutions. A user of a construction kit is in the situation of a chess novice who knows the goal of the game and sees all the different pieces on the board; this user, however, is far from being able to play a good game. Our notion of *design environments* is intended to also support the semantic and pragmatic levels. A design environment has knowledge about the function of the components and how they can be used to achieve higher-level goals.

In this article, we first articulate some problems in human-computer communication. We describe different approaches and discuss specific challenges and unique opportunities that knowledge-based systems create for human-computer communication (Section 2). In Section 3, the notion of

human problem-domain communication is elaborated. In Section 4, general characteristics and a number of examples of construction kits will be presented. The next section (Section 5) is concerned with two systems that represent steps toward the goal of creating knowledge-based design environments. Finally, we discuss our experiences and 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, 1983).

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. High-functionality computer systems (containing a wealth of information) offer great possibilities but, at the same time, they pose a large number of problems. Some of the requirements designers have to address are:

1. Help break the complexity barrier, for example, by supporting user- and task-specific filters and dynamic unfolding, that is, showing parts of the system only when they are relevant.
2. Help break the utility barrier, defined as the ratio of value to effort expended; this can be done either by increasing the value of a system or by decreasing the effort needed to learn and use it.
3. Give control to users when they need or desire it; do things automatically when users do not want to be bothered.
4. Support active exploration, for example, by undo and redo mechanisms.
5. 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.

6. Take advantage of modern hardware and basic software capabilities; for example, use screens as a two-dimensional world that can be edited (direct manipulation).

One of the most promising approaches to cope with these requirements are knowledge-based systems that contain knowledge about specific problem domains, the communication partner, the communication process (e.g., recovery from breakdowns), and help and explanation facilities to increase their comprehensibility. They do not require that all information has to be communicated explicitly.

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 issues of reliability, comprehensibility, and trust in the system's performance become more important (Chambers & Nagel, 1985).

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 such as in the MYCIN system (Buchanan & Shortliffe, 1984). These dialogs provide little help in problem definition. Frequently, however, users learn incrementally about the nature of their problems, and they want to solve them in cooperation with a system (Woods, 1986). This requires better communication capabilities than most systems traditionally have offered.

Failed attempts to build fully automatic systems (e.g., automatic programming or high-quality machine translation systems; Winograd & Flores, 1986) have shown that for many domains, a symbiotic, cooperative system architecture is more adequate and promises greater success than an automous 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. In partially 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.

To overcome some of these problems, we identified the following design constraints for human–computer systems:

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 (see Chambers & Nagel, 1985): 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 for graphical interfaces (described in Section 4.2). In the domain of software engineering, we have built components of a software oscilloscope that, in analogy to an electronic oscilloscope, visualizes static and dynamic aspects of programs (Boecker, Fischer, & Nieper, 1986).

These observations provide the rationale for our major research area: how to bring knowledge-based systems and human-computer communication together to construct systems that are useful and usable. The design constraints just enumerated can be used to provide some global guidance for the construction of better human-computer communication and have played a crucial role in the development of the systems described in this article. A drawback is that they are not prescriptive enough to indicate how one should proceed within the context of a specific system design.

2.2. Existing Interface Construction Techniques

In order to solve these problems, different interface construction techniques have been proposed.

Natural Language Front-Ends. Because humans communicate with each other quite easily using natural language, it is a natural step to study the

applicability of natural language to the human-computer communication problem. 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, 1984). 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, 1984; Robertson, McCracken, & Newell, 1981). In natural language interfaces, the computer is the listener and the human is 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 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.

Use of a natural language front-end implies that appropriate interactive behavior can be achieved by tacking an off-the-shelf natural language front-end onto an existing system. This is a fallacy. Many human-computer systems 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 elaborate data models and knowledge representations (Williams, Tou, Fikes, Henderson, & Malone, 1982) 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, 1986). Assuming we had a natural language interface to UNIX (Wilensky, Arens, & Chin, 1984), 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, the same resources are not available for their detection and repair.

User Interface Management Systems (UIMS). UIMSs (Olsen et al., 1984) provide graphic primitives and tools for specifying dialogue structures (ATNs, context free languages). By providing a uniform set of high-quality primitives, UIMSs attempt to foster the construction of consistent interfaces that can be rapidly developed. Most UIMSs are based on a strong separation of interface and application code. This is a good approach to problems for which there is

only a limited information exchange. 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.

3. HUMAN PROBLEM-DOMAIN COMMUNICATION

Most computer users are not interested in computers per se, but they want to use the computer as a tool 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. The important abstract operations and objects of a given application area are directly built into the environment. This implies that the user can operate with personally meaningful abstractions, and learning processes are reduced by exploiting the user's knowledge of the problem domain.

For a few applications and a few users, a predefined set of functions will suffice to accomplish nearly all tasks. For most applications, though, users need to customize and modify the behavior of the tools that they use in order to solve their particular problems.

Traditionally, this could only be done by programmers who understood how abstractions of the application domain were implemented with computational methods. The goal of human problem-domain communication is to remove the distinction between programmers and users. This can be achieved by representing a more complete model of the application domain. Specifying the desired behavior of the tool can, in most cases, be achieved without having to resort to programming language concepts like for loops and if then else statements. Many application domains have natural ways of expressing control; see, for example, repeat signs in musical notation or the laws of motion in the domain of physics. Instead of specifying conditionals and loops, human problem-domain communication means combining the domain building blocks so that they function as desired. What must be done is to move away from programming languages, even domain specific ones, and move toward providing a set of programming abstractions that are within the users' application areas.

A similar shift can be seen in architectural design in the work of Alexander (1964). In his early book, *Synthesis of Form,* he described a mathematical framework for design whereas in his later book, *A Pattern Language* (Alexander et al., 1977), he articulated a set of patterns that can be used and understood by the people involved and affected by the design process (Hooper, 1986).

3.1. Modeling Problem Domains

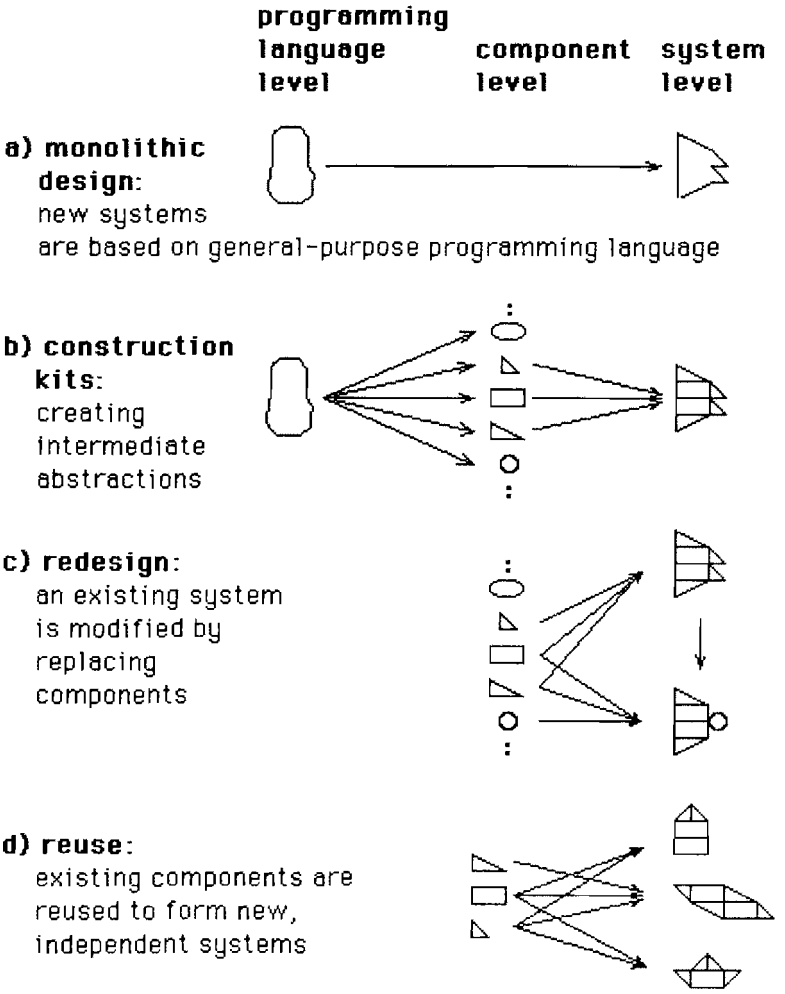
Human problem-domain communication requires environments that support design methodologies whose main activity is not the generation of new, independent programs, but the integration, modification, and explanation of existing ones (Winograd, 1979). Just as one relies on already established theorems in a new mathematical proof, new systems should be built as much as possible using existing parts.

Many large software systems, however, are built as monolithic systems, directly implemented on top of a general purpose programming language (Figure 1-a). Although these systems are structured in some way, this structure usually does not correspond to established abstractions of the application domain. With construction kits and design environments the latter approach is taken (Figure 1-b). These systems provide one or more intermediate levels of problem-oriented building blocks. The existence of these intermediate substrates enables users to redesign and adapt their systems (i.e., modify the original system; Figure 1-c) as well as to reuse existing abstractions to form new systems (Figure 1-d; Fischer, 1987a; Fischer, Lemke, & Rathke, 1987). 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: How well does the user have to understand existing components?

The existence of the right components is not enough; they must be assembled in some way to form a well functioning whole. There are combination operators of different complexity (Fischer & Lemke, 1988). Fitting programs together using the UNIX pipe mechanism is an example of simple combination. Filling in forms can be viewed as instantiation of a template.

The object-oriented paradigm has emerged as a powerful and easy to understand structuring method. Objects encapsulate procedures and data. They represent stable intermediate parts that, as Simon (1981) demonstrated, led to much faster evolution of complex systems. Objects are grouped into classes, and classes are combined in an inheritance hierarchy. This inheritance supports differential design (i.e., object y is like object x except $u,v,$ . . .). Object-oriented formalisms support design by instantiation as well as design by specialization through the creation of subclasses. Subclasses can inherit large amounts of information from their superclasses, and new objects that are almost like other objects can 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. The creation of subclasses is more complex than instantiation, but also more powerful because the behavior of the superclasses

Figure 1. Modeling problem domains with application-oriented abstractions.



can be augmented and overwritten in arbitrary ways. Many tasks can be achieved before one has to use the full generality of subclassing.

3.2. Intelligent Support Systems

Systems that make an attempt to model many different problem domains will be large and complex in order to provide all the necessary abstractions. The CommonLisp standard, for instance, specifies more than 600 functions. More comprehensive systems, like UNIX or LISP machines, provide a larger

number of abstractions by far. This richness 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, 1987b) indicate that the following problems prevent designers from successfully exploiting the potential of high-functionality systems:

1. Designers do not know about the existence of needed objects (either building blocks or tools).
2. Designers do not know how to access objects.
3. Designers do not know when to use these objects.
4. Designers do not understand the results objects produce for them.
5. Designers cannot combine, adapt, and modify objects for their specific needs.

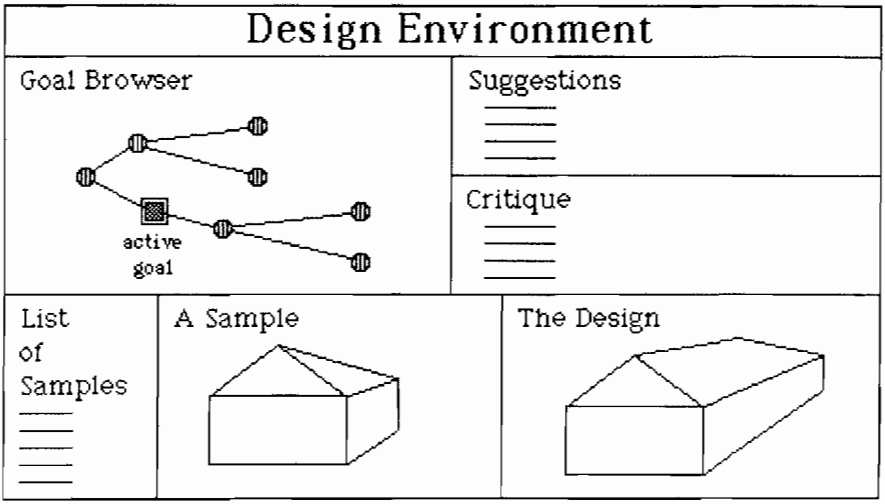
Unless we are able to solve these problems, designers will constantly reinvent the wheel instead of taking advantage of already existing tools.

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 have shown that, on the average, only a small fraction of the functionality of complex systems such as UNIX, EMACS, or LISP is used (Draper, 1984; Fischer, Lemke, & Schwab, 1985).

In the early days of computing, programs consisted of a number of algorithms on punched cards. Interactive systems emphasized the importance of the user interface. For the just described high-functionality computer systems, simple interactive user interfaces are no longer sufficient, and intelligent human-computer communication facilities are required. We have constructed a number of intelligent support systems including documentation systems (Fischer & Schneider, 1984), help systems (Fischer et al., 1985), visualization components (Boecker et al., 1986), and critics (Fischer, 1987b).

In the past, these intelligent support systems have been constructed as isolated components. We are in the process of combining them into an integrated design environment whose architecture is shown in Figure 2. The

Figure 2. **An architecture for an integrated intelligent design environment. The picture shows components such as the problem decomposition goal tree, suggestor, and critic. These components contain design knowledge for supporting design processes by the user.**



following components represent knowledge about the problem domain as well as about the design process:

1. Hierarchical problem decomposition: An important part of design knowledge is knowledge about how the problem can be attacked, which subtasks have to be solved, and how partial designs can be composed.
2. Critics: Critics are system components that recognize possible improvements, trouble spots, inconsistencies, or even just breaks in style. For the domain of programming in LISP, we have developed an intelligent support system that criticizes code with the goals of making it more cognitively or more machine efficient (Fischer, 1987b). The domain of general LISP programming is relatively unstructured. A critic can be more powerful in semantically richer domains.
3. Suggestions: Suggestions can be content-oriented and organizational. Content-oriented suggestions can say something about how to improve the design and how to solve problems. Organizational suggestions can help in deciding what to do next.
4. Animated demonstrations: One of the most powerful tools to understand the dynamic aspects of a system is a demonstration.
5. Samples: Representative examples play an important role in design disciplines such as architecture. Samples of different types of designs are another way to represent design knowledge.

4. CONSTRUCTION KITS

A construction kit is a set of building blocks that models a problem domain. The building blocks define a design space, that is, the set of all possible designs that can be created by combining these blocks. In the following sections, we briefly describe some examples and indicate how the problem of adding new elements to a construction kit can be addressed.

4.1. The Pinball and Music Construction Kits

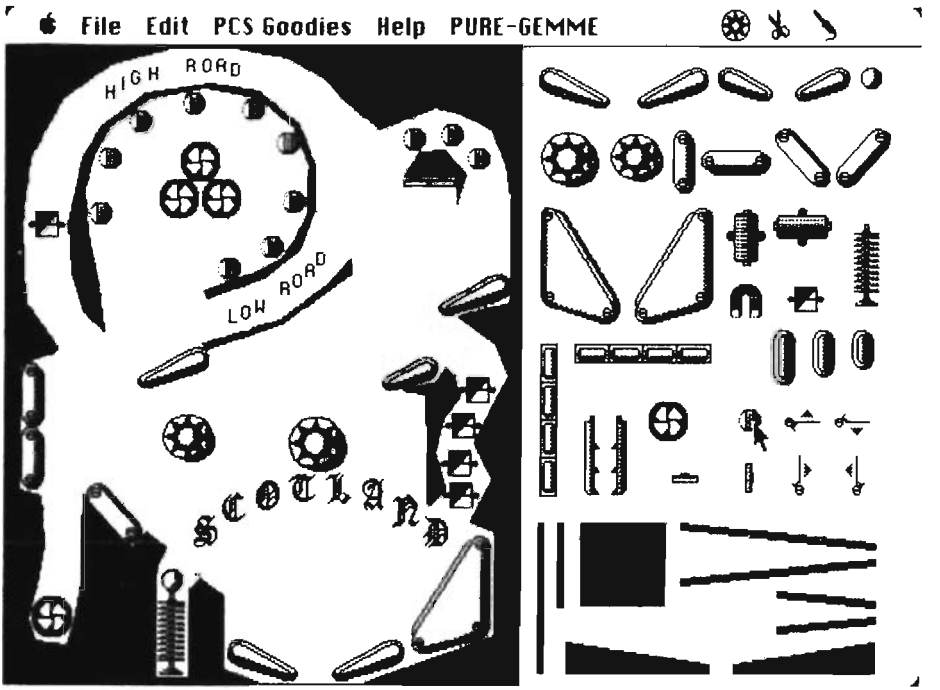
The Pinball and Music Construction Kits (two interesting programs for the Apple Macintosh from Electronic Arts; see Figure 3) provide domain-level building blocks (e.g., bumpers and flippers; staves, piano keyboard, notes, sharps, etc.) to build artifacts in the two domains of pinball machines and musical composition. Users can interact with the system in terms with which they are already familiar; they need not learn abstractions peculiar to a computer system.

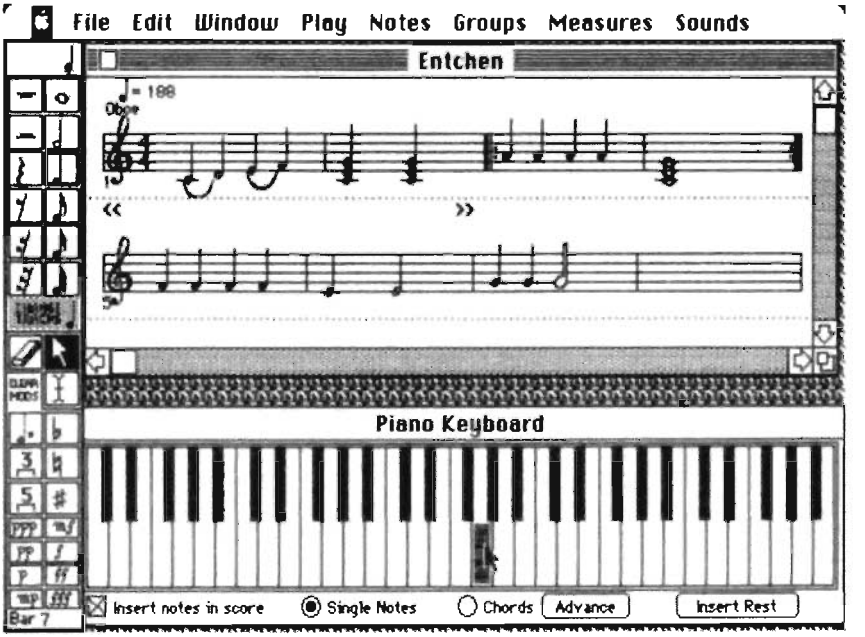
Our empirical investigations have shown that these systems come close (within their scope) to our notion of human problem-domain communication. Users familiar with the problem domains but inexperienced with computers had few problems using these systems, whereas computer experts unfamiliar with the problem 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. By using the construction kits, our subjects had a sense of accomplishment because they were creating their own impressive version of something that works, yet is not difficult to make.

Persons using the systems do programming, but the programming consists of constructing artifacts in the domain and not of writing statements of a general-purpose programming language. This kind of programming is comparable to writing a text using a document formatter such as TEX or SCRIBE. It is a process of creating a specification that is interpreted by a document processor and printer or by the run-time systems of the kits to direct the rolling ball or to generate sound. Using a WYSIWYG editor, however, is different and does not qualify as programming because the final product is created directly and there is no step of interpretation. The created document is virtually identical to hardcopy generated from it.

By evaluating the Pinball and Music Construction Kits as prototypical examples against our objective to support human problem-domain communication, we have identified some shortcomings. The two systems do 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

Figure 3. Screen images from the Pinball and Music Construction Kits.





Pinball Construction Kit allows users to build sets in which balls get stuck in certain corners and certain devices may not be reachable (Hutchins, Hollan, & Norman, 1986). To assist users in constructing truly interesting objects, more powerful design environments are needed.

For almost any domain, the set of domain abstractions is not clearly defined or evolves over time; the number of objects at the component level of Figure 1 is not fixed. Therefore, there is a need to modify and extend the set of existing abstractions. Here the Pinball and Music Construction Kits fall short. The various EMACS editors and the ZOO system (Section 4.4), however, are examples of systems that provide extension mechanisms (multiple component levels).

For these construction kits, the physical metaphor consisting of spatial organization and simple combination of parts (e.g., associating sound with a bumper) has proved very powerful. If one considers the Pinball and Music Construction Kits as success models, then one has to investigate whether the physical metaphor can be generalized to other problem domains or if there are other, equally intuitive metaphors.

4.2. WLISP: A Construction Kit for User Interfaces

Over the last several years, we have developed WLISP (Boecker, Fabian, & Lemke, 1985; Fabian, 1986), an object-oriented construction kit for human–computer communication, and a large number of associated tools and intelligent support systems for exploiting this kit effectively (Figure 4). The WLISP building blocks are organized as inheritance networks in an object-oriented architecture based on the ObjTalk language (Rathke, 1986). This architecture provides for components on multiple levels and facilitates the extension of the set of available blocks. Much more so than in the domain of pinball machine design, it was (and still is to some extent) unclear what the right abstractions in user interface design are. Our experience indicates that the development of the right abstractions (and their embedding into inheritance hierarchies) is a difficult process that takes time and has to proceed in an evolutionary fashion driven by the development of application systems that are based on these abstractions. Currently, there are over 200 classes representing abstractions about different kinds of windows such as superwindows, paned windows, menus, icons, gauges, and so on. The inheritance network is still changing, thus indicating our growing understanding about the domain of two-dimensional interfaces.

The example of a graphical UNIX directory browser demonstrates the use of object-oriented components of the WLISP construction kit. Figure 5 shows a browser with four directories (andreas, Lisp, kbpe, tristan-kit) and two plain files (tristan-kit.l, tools.l). The user can selectively display various parts of the global file hierarchy and execute actions on the displayed files.

Figure 4. The WLISP programming environment. A number of systems constructed using the WLISP construction kit are shown. Among others, there is a file system directory display entitled **boulder**; it is implemented as a subclass of the static menu class. The **Wlisp RC Sheet** at the bottom left is an editor for system parameters (Fischer & Lemke, 1988); it is based on components for electronic forms. The kaestle window on the right is a graphical editor for LISP data structures (Boecker et al., 1986).

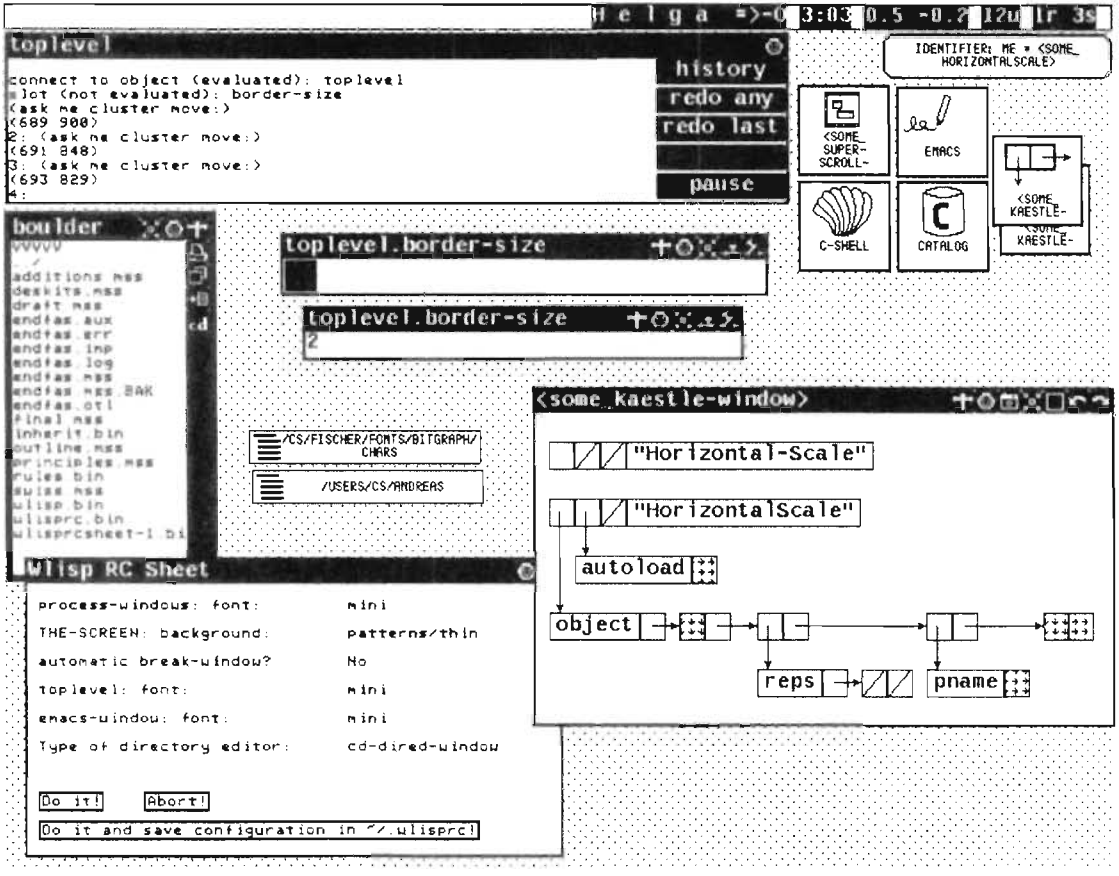


Figure 5. **A hierarchical UNIX directory browser.**

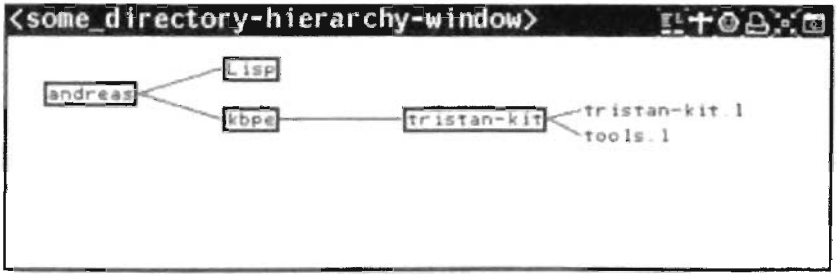


Figure 6 illustrates the component structure of the UNIX directory browser. The files and directories are represented as instances of class unix-file-node and unix-directory-node, respectively. These two classes were explicitly created for this application. They inherit from multiple superclasses, which are parts of the WLISP kit. File nodes are based on the class adaptive-text-region, which displays a string of text in a rectangular area whose size is automatically adapted to the size of the text. Directory nodes use adaptive-text-region-with-border instead, which in addition draws a border around the text. Common to both classes are the superclasses node-mixin, node-repr-mixin, and node-util-mixin. These superclasses provide the functionality for links between nodes, selective display of parts of a graph, graph editing, and such. The browser window itself (class unix-directory-window) is a subclass of other predefined classes that provide functionality pertaining to the directory graph as a whole (e.g., automatic layout planning).

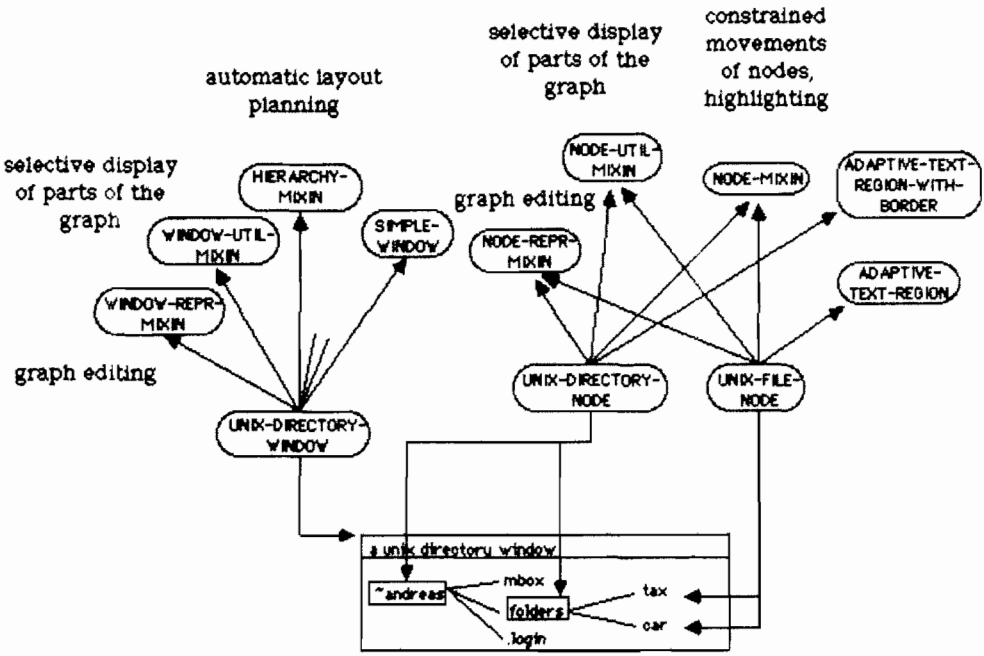
By using WLISP, the "human–computer communication design question" is answered by providing appropriate building blocks that suggest good designs. The object-oriented system architecture is highly flexible and enhances the reusability of many building blocks. In creating new human–computer communication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

For the large set of components of WLISP, the problems described in Section 3.2 became prominent. Where the Pinball Construction Kit has in the order of tens of components, a general purpose user interface kit easily grows to many hundreds or thousands. For such a system, displaying a simple palette with all components is no longer feasible.

4.3. FINANZ: A Financial Planning Kit

FINANZ (Rathke, 1986) is an advanced financial planning system extending the spreadsheet paradigm (see Figure 7). Spreadsheets have become

Figure 6. **The component structure of the UNIX directory browser.**

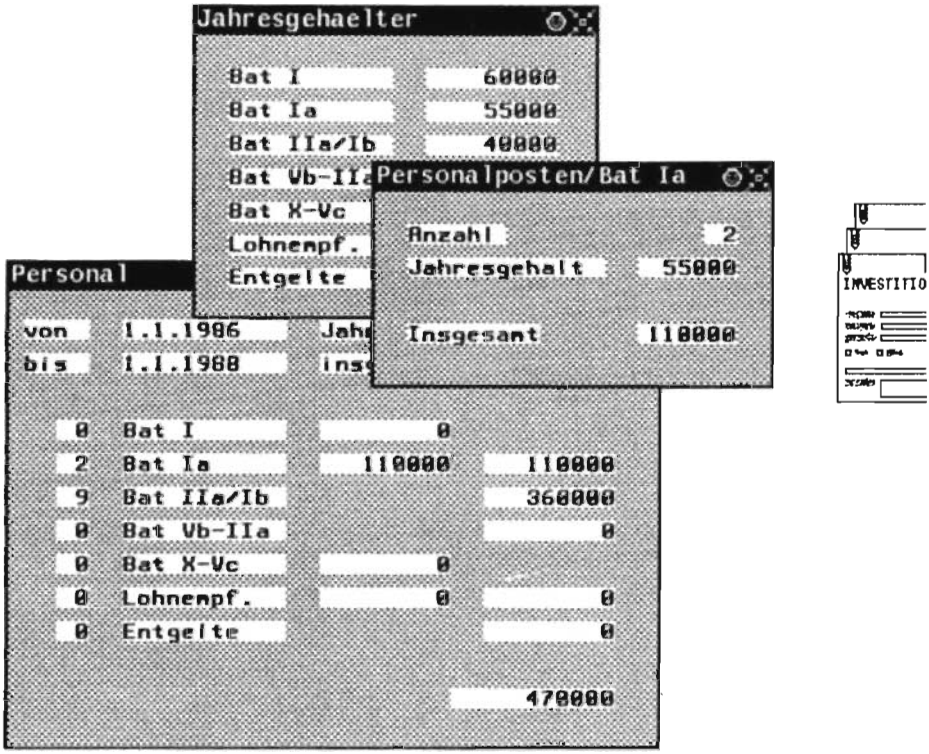


success models for computer systems not because they are "smart" programs, but because they let users operate in a systematic domain that is directly relevant to their work. FINANZ differs from ordinary spreadsheet programs in that the relationships among the form fields are represented by internal knowledge structures that model the knowledge of the application domain.

The main characteristics of FINANZ are (for details, see Rathke, 1986):

1. In its basic configuration it can be used as a regular spreadsheet system. From there it can be gradually augmented to a knowledge-based system without losing its basic supportive style of interaction.
2. 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.
3. 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 that can be modified to serve the needs of the application.
4. The 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.

Figure 7. **The financial planning system FINANZ.**

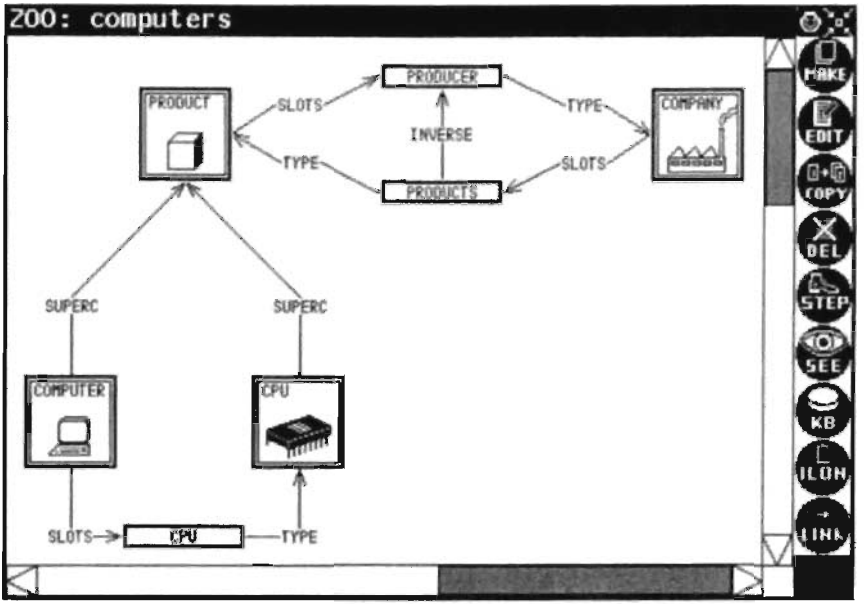


The capabilities of the system are based on ObjTalk (Rathke, 1986). Concepts about the application domain, the user, and the dialogue are represented as active objects that communicate by message passing. Their behavior is described in classes that form a hierarchy among which knowledge is inherited. 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. FINANZ is a system building effort to provide a substantial amount of flexibility and tailorability to end-users without requiring that they become programming experts.

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

In Figure 1, the redesign and reuse processes are based on the assumption that the needed components are available in the construction kit. But what

Figure 8. **The knowledge editor ZOO.**



happens if these elements do not exist? Obviously the user who is familiar with the underlying programming language can go back and define the required component at this level. But assuming that the user can do so misses the whole point of our system building effort: to protect the user from the complexities of the lower levels. What is needed here is a middle ground between powerful but specialized construction kits and general purpose low-level programming languages.

One solution to avoid this problem is that the construction kit designer provides a complete set of building blocks. But for ill-structured problem domains where no formal description of the problem space exists, this option remains wishful thinking. Another way to approach the problem is to provide high-level interaction techniques and metasystems that support the construction of new building blocks without the necessity to descent to low levels.

ZOO (Riekert, 1986), implemented in WLISP and ObjTalk, provides graphical support (Figure 8) for constructing new domain-dependent abstractions without being forced to go down to the ObjTalk or even the LISP level (a possibility that 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.

The graphic representation provides two kinds of graphic primitives: icons and labeled arrows. Icons are used to represent objects, and 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. Figure 8 displays the knowledge that the class of computers and the class of CPUs are both products, products are produced by companies, and companies produce products (the inverse relationship that can be generated automatically). ZOO can be used both as an instrument for inspecting the contents of a knowledge base and as a tool for modifying and augmenting a knowledge base by direct manipulation of sensitive screen objects. When the user creates graphical objects on the screen, ZOO internally generates descriptions of ObjTalk objects. From the perspective of knowledge-based systems, it is a tool to address the problem of knowledge acquisition.

5. DESIGN ENVIRONMENTS

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 for efficient user interface design. The existence of good user interface components, however, does not guarantee 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 that 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 environments 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 environments 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 allow to run the resulting system, design environments 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 environments constrain the problem space, leaving beginners with fewer choices by providing defaults and grouping the available functions.

Design environments 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 environments for specific areas of the WLISP construction kit. WIDES is a design environment for basic characteristics of window types, and TRIKIT is a design environment for graph display and edit tools.

5.1. WIDES: A Window Design Environment

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 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: (a) to provide a level of abstraction above the object-oriented implementation of these components, (b) to reduce the knowledge required to use the components, (c) to make their use more effective by preventing errors and suggesting the right components to use, and (d) 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 environment allows its users to create specific window types for their applications.

In the following sections, we give an example of how WIDES employs techniques like menu selection and an adaptive, dynamic suggestion list to greatly simplify window design. Merits and shortcomings of WIDES are discussed.

Description of WIDES. The initial state of the system is shown in Figure 9. It is a window with four panes: (a) a code pane that displays the current definition of the window type, (b) a menu of suggestions for enhancements of the window type, (c) a history list, and (d) a menu of general operations.

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 10); 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 11. Two

Figure 9. Initial state of WIDES.

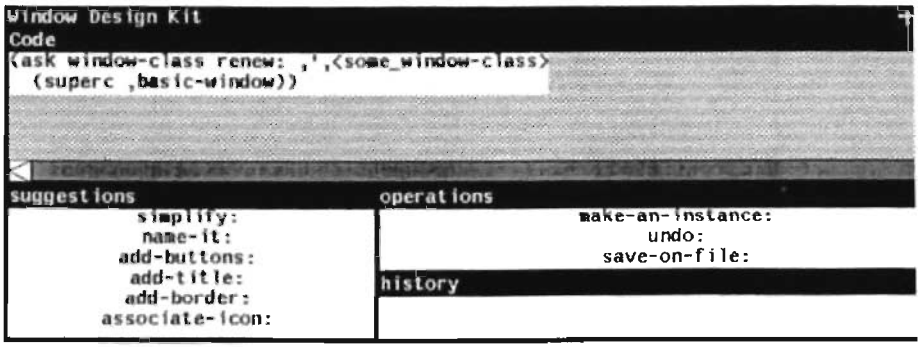
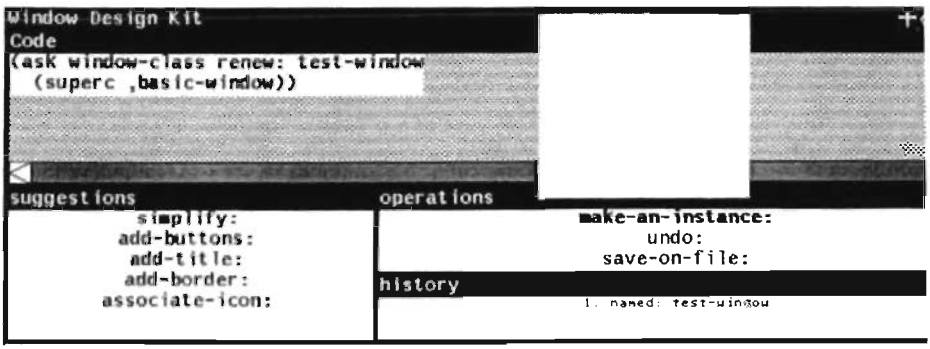


Figure 10. WIDES: An instance of the current window definition has been created.



superclasses, border-mixin and title-mixin, have been added to the definition; a new instance shows a 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.

The next figure (Figure 12) 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 13 shows that the input has been added as a default for the title slot.

Figure 11. **WIDES: Title and border have been added to the window type.**

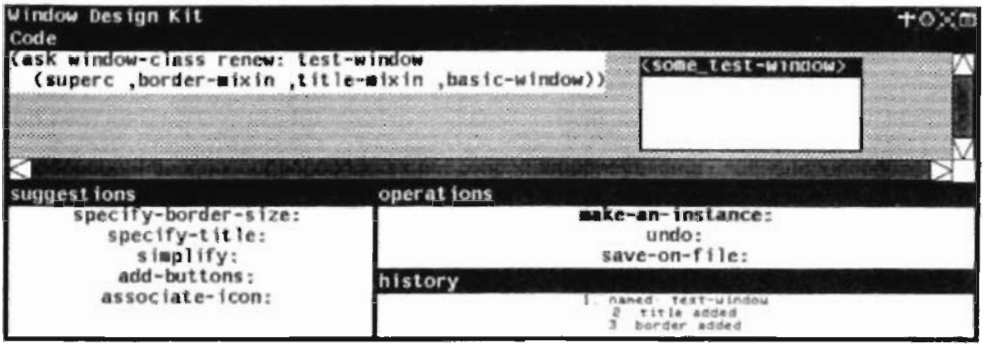
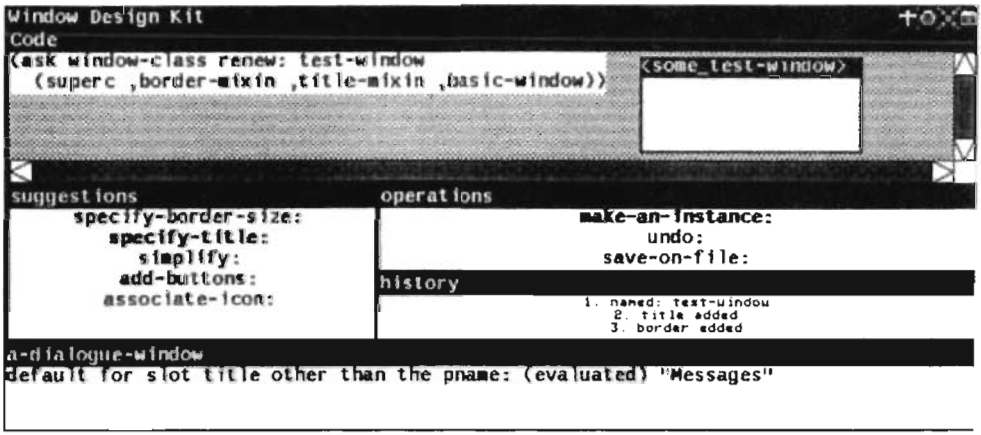


Figure 12. **WIDES: Specification of the title.**



If specific inputs are required, we cannot expect the user to know what the legal inputs are. Therefore, as shown in Figure 13 in which the user associates the window with an icon, a menu of alternatives is displayed (see the pop-up menu at the bottom left). Figure 14 shows a window and an icon of the selected type (document-icon).

An even more complex modification is demonstrated in Figure 15. 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

Figure 13. **WIDES: Association of a different icon to the window.**

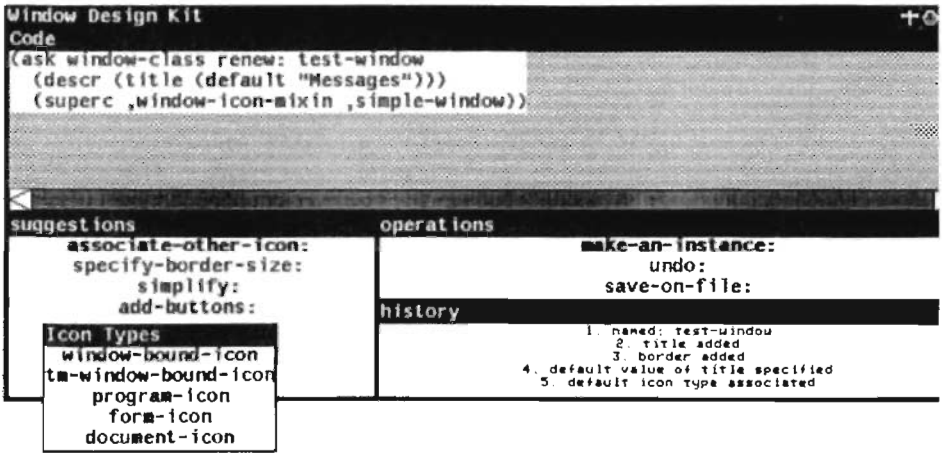
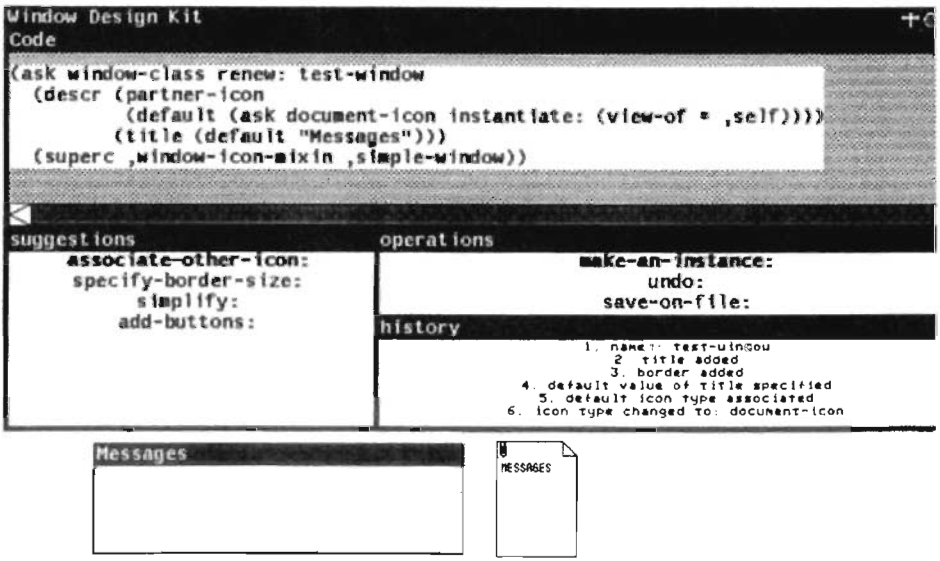


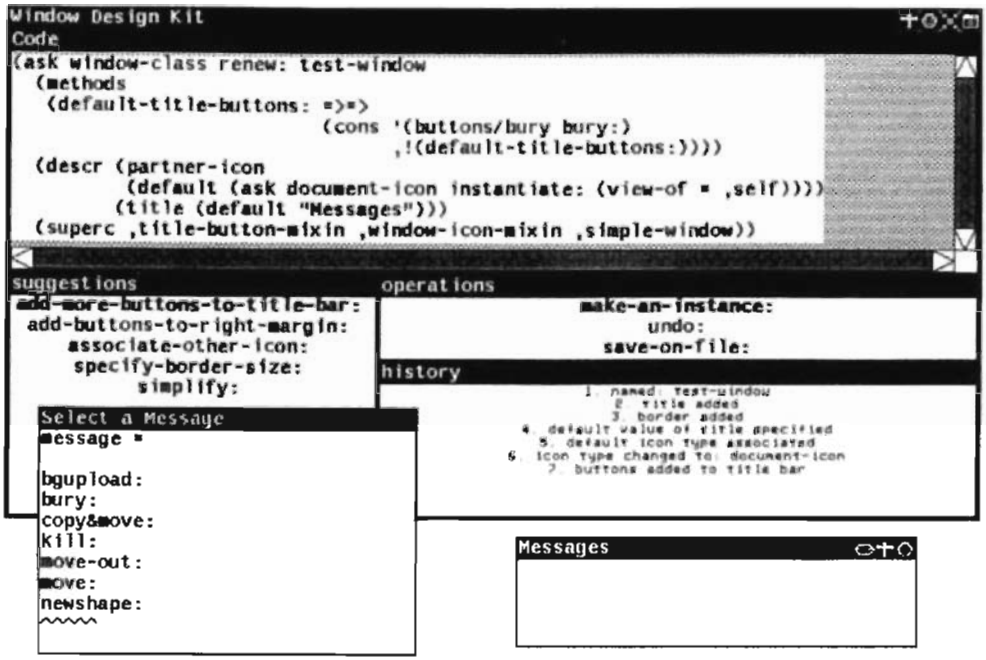
Figure 14. **WIDES: The window and its associated icon.**



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.

Figure 15. **WIDES: Adding a button to the title bar.**



Discussion. In order to construct a new window type, users no longer need to know what building blocks (*superclasses*: e.g., title-mixin) exist, what their names are, and how they are applied, or 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. Not having to directly write the code as displayed in the code pane is a significant advantage for the user.

But, in addition to supporting the translation of a specification into working code (as is done by many other code-generating systems), WIDES also supports the specification process itself. An unexperienced user who is discriminately presented with the large number of design decisions that can be made for the design of a window is very likely to fail to produce a reasonable design. An important capability of experts is knowing the critical parts, those parts that influence most other parts.

In WIDES, the task of window design is decomposed into a hierarchy of semantically meaningful subtasks. The critical decisions form the highest levels, and the dependent tasks can be found at the lower levels and the leaves. The suggestions menu is dynamically updated according to this model of window design. Suggestions, such as adding a button to the title bar, are given when appropriate within the overall design process. On the other hand, the

dialog is not completely system-driven and the user can direct the design process towards desired aspects of the design.

WIDES does not support direct editing of the generated code in the code pane. Analyzing arbitrary program code is generally hard because the descriptive power of general program code is greater than the one of 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 below the domain level are impossible—that the system is internally consistent and runnable. Because the system cannot have knowledge about arbitrary uses of windows, it cannot prevent the designer from specifying a wrong window title such as a title that would be confusing to read.

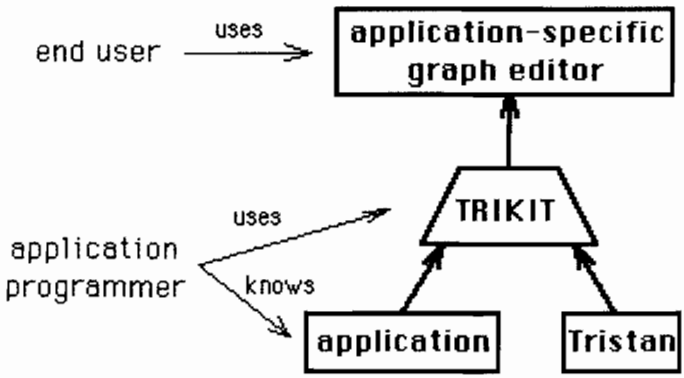
Because the system can guide the window designer only to a certain level (though this level is higher than with construction kits), it supports a prototyping design style. The undo: operation makes it easy to step back and retract a decision in order to proceed differently. So far, the UNDO feature has 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 also have to be removed.

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 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 that has to be augmented by more ObjTalk code. Still, users (students in our department) found it exciting that, with some menu

Figure 16. **Usage of TRIKIT.**



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: An Environment for the Design of Application-Specific Graph Editors

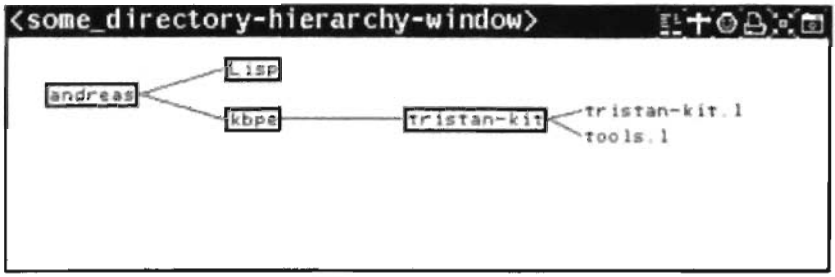
A very common user interface problem is the display and modification of hierarchical and network structures. Application systems that deal with rule dependency graphs, concepts of a domain of expertise (for explanation purposes), goal trees, or inheritance hierarchies in object-oriented languages are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

Our response to this problem was to build a design environment for graph display and edit tools. Figure 16 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: (a) ObjTalk inheritance hierarchy editor, (b) UNIX directory browser (see Figure 17), (c) subwindow hierarchy display, (d) a project team hierarchy, and (e) EMYCIN rule dependency display.

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 use the example of a hierarchical directory browser, which

Figure 17. **An example application of TRIKIT.**



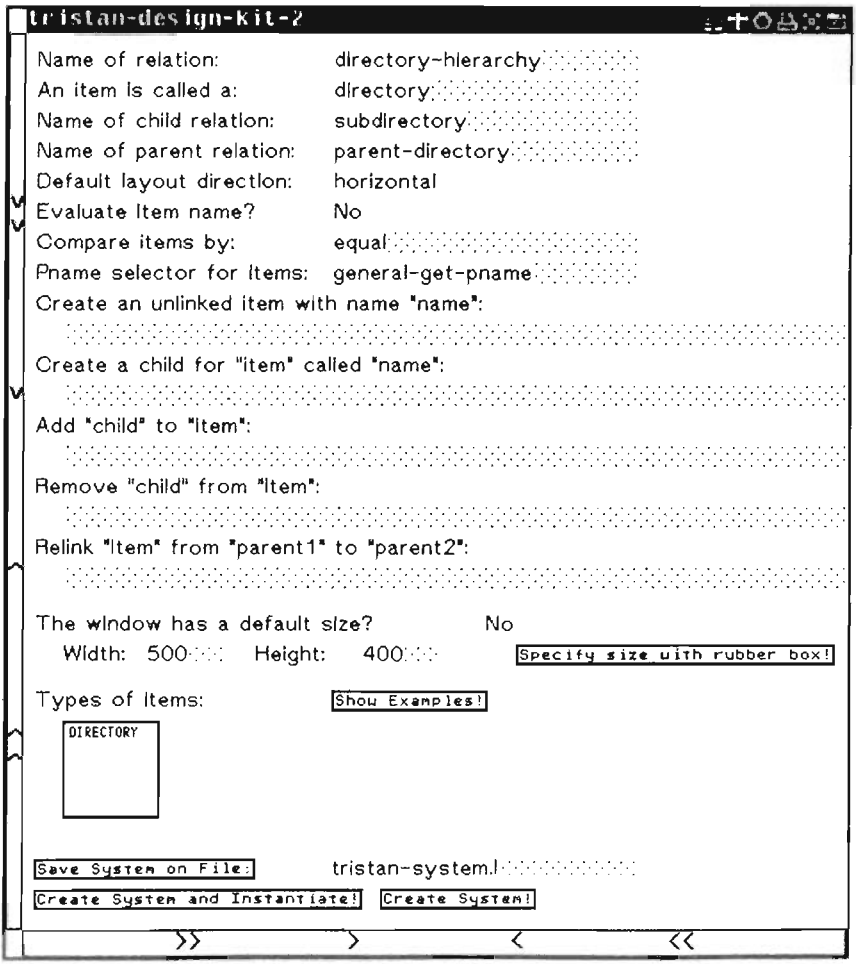
may be displayed as in Figure 17 (repeated from Figure 5). Here the nodes are directories and files that are the leaf nodes. The lines represent the membership relation between files and directories, which can also be members of other directories. Each of the nodes is a data structure with properties like name, creation time, owner, protection, and size. There are operations to retrieve pieces of the graph (e.g., list-directory) and to create and delete nodes and lines.

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 17).

The Generic Graph Display/Editor. TRIKIT is based on TRISTAN (Nieper, 1985), a facility of WLISP for building direct manipulation display and editing systems for graph structures. TRISTAN supports the following operations: (a) 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); (b) automatic layout planning; (c) manual layout modification by constrained moving of nodes; (d) highlighting of nodes; and (e) 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 18 (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.

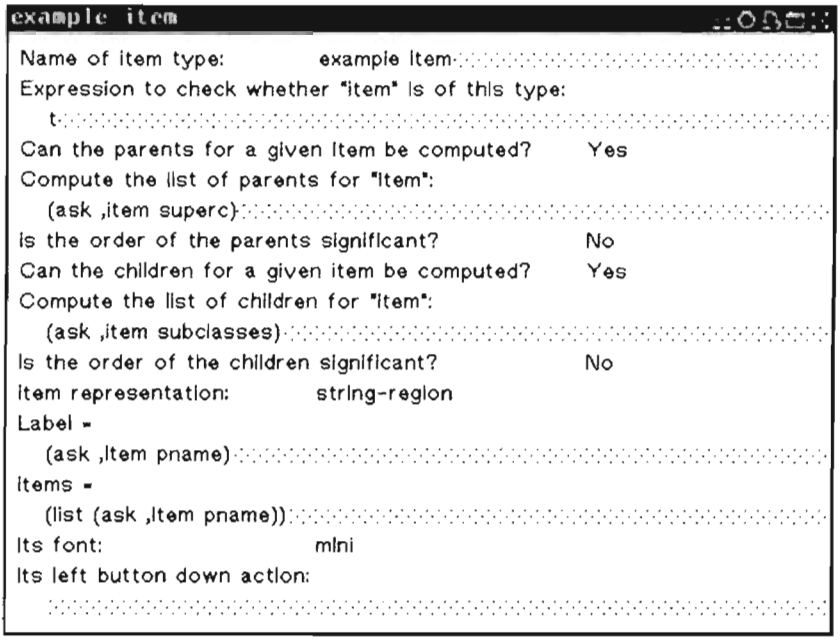
Figure 18. **Initial state of the main form and an inheritance hierarchy window generated from it.**



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, and so on; 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.

Figure 19. **Initial state of the node form.**



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—an ObjTalk inheritance hierarchy display system (Figure 18, bottom window). This allows users to familiarize themselves with TRIKIT, to modify parameters, and to learn their significance.

Clicking the square representing the example item subform produces the form of Figure 19. The main form is associated with the graph in general, but 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 17. 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 20 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

Figure 20. **Main form modified to describe a directory editor.**

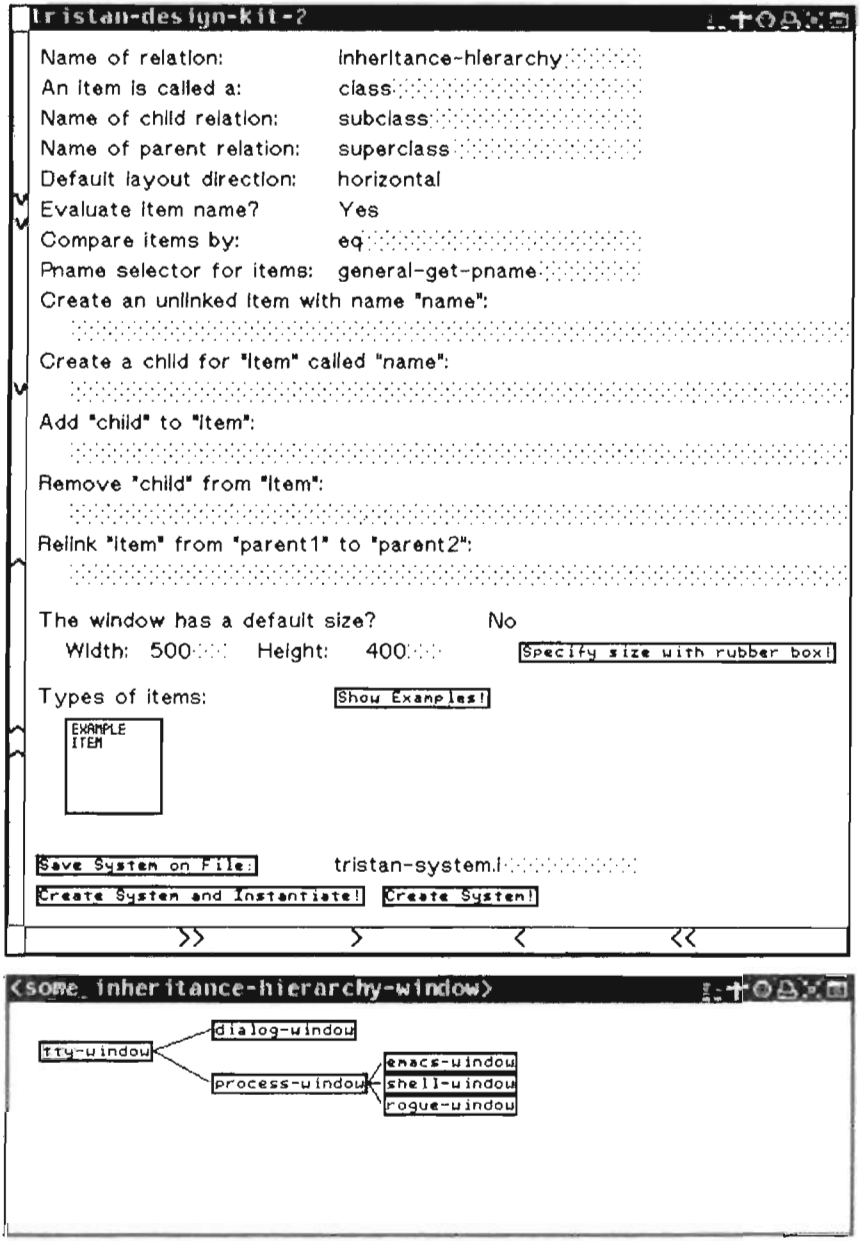
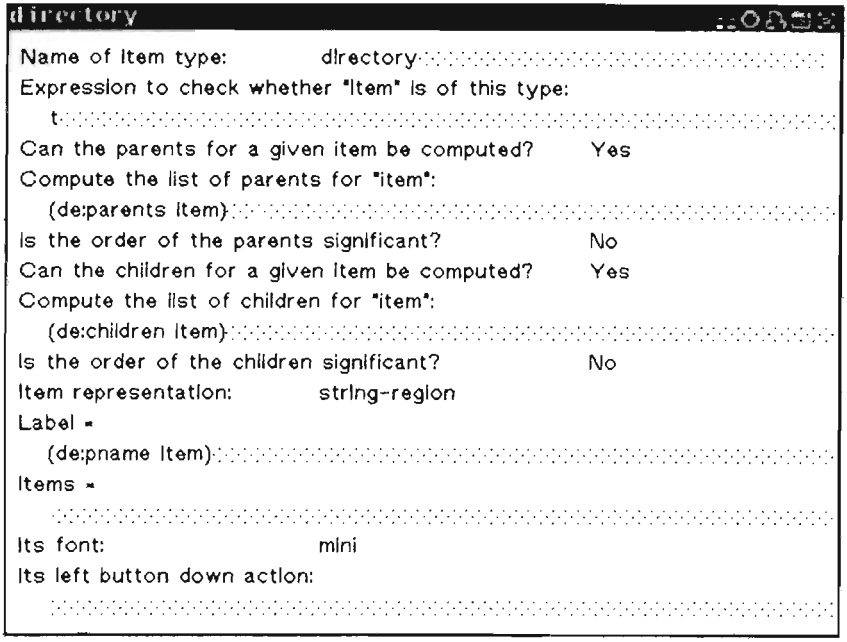


Figure 21. **Node form modified to describe a directory node.**



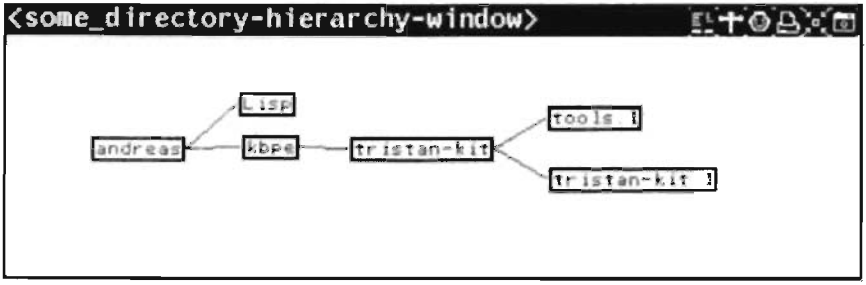
system. They describe the names of the relation to be displayed, the names of the items that are elements of the relation, and those of the links to superordinate and subordinate nodes in the relation. The Evaluate item name? field says that a user-entered name of a file or directory represents itself 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 21), 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.

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

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

Figure 22. **An example directory hierarchy window.**



of it is the directory-hierarchy window type, which is being instantiated to produce the window of Figure 22. (The "Create System" push button just recompiles the forms, but does not create any new instances.)

Figure 23 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:

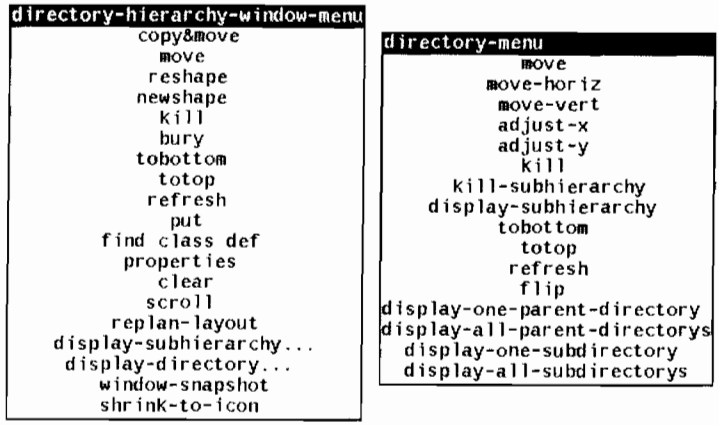
replan-layout	supplied by the TRISTAN system: automatically rearranges the layout of the graph using a general planning algorithm;
display-subhierarchy . . .	also supplied by the TRISTAN system: displays all the recursively subordinate nodes of a given node;
display-directory . . .	generated by TRIKIT: displays a given directory. This is a renamed version of the generic display-item . . . operation of TRISTAN.

The menu to the right of Figure 23 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.

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-child and de:move (Figure 24).

Also, there are actually two types of nodes in the application: directories

Figure 23. **The main menus of the directory hierarchy window and the directory node.**



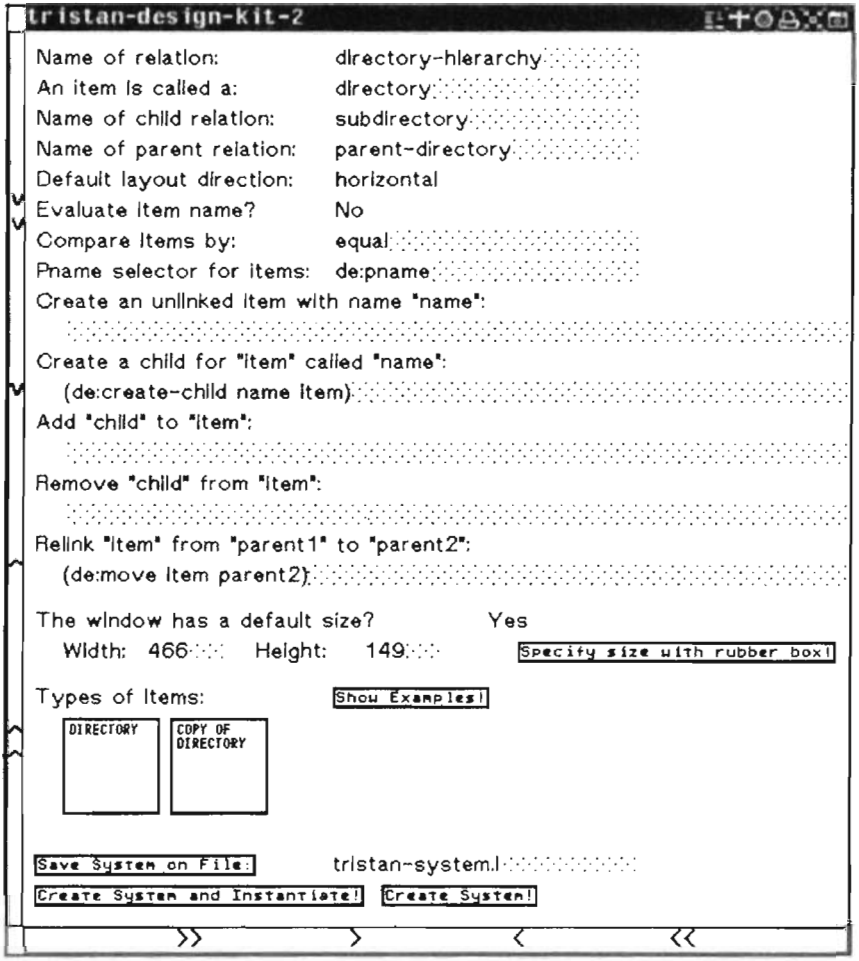
and files. The user creates a new subform (Copy of Directory) by cloning the existing one. Now the system must be told how to distinguish between the two node types. Consider Figure 25, in which the second field specifies the necessary predicate expression (de:directoryp). There has also been an action 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 26). 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 17.

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.

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 vs. vertical layout of the graph), not on the implementation level (ObjTalk classes). At the interface between graph editor and application, code-level specifications have to be used such as the code that computes the list of parents for a file item ((de:parents item), see Figure 21). This form of specification does not pertain to the graph as such, but is required to be independent of the internal representation of the graph in the application system.

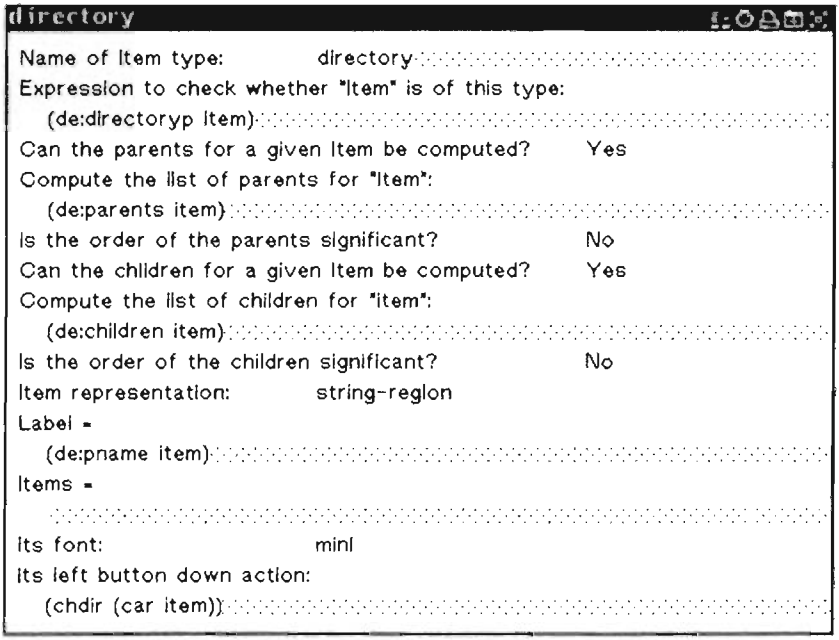
Figure 24. **Extended description of the directory hierarchy window.**



A critical design decision was whether to make users 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 because we did not find clear and intuitive terms for the different concepts and because there was no need to be so specific, we dropped this distinction from the model of the design process that is presented to the user. This is an example of the abstraction from implementation details that we hope to achieve with design environments 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 first version, which may

Figure 25. **The directory node form.**



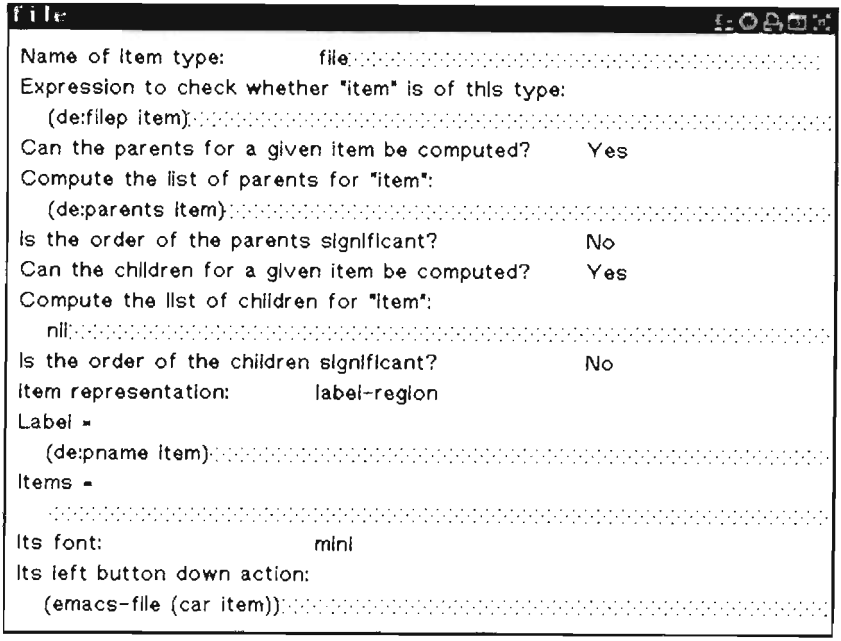
be refined on a lower level. Additional knowledge about graph data structures can further enhance the power of the TRIKIT design environment. Given an example of a graph, or a description of the data structure such as a type definition, the system could automatically produce an initial display design.

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 imperative 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 should be an additional alternative for the expert, and the suggestion list should be kept because it 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.

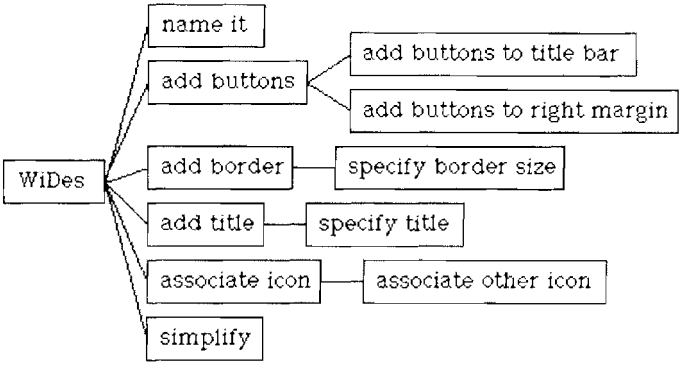
Figure 26. **The plain file node form.**

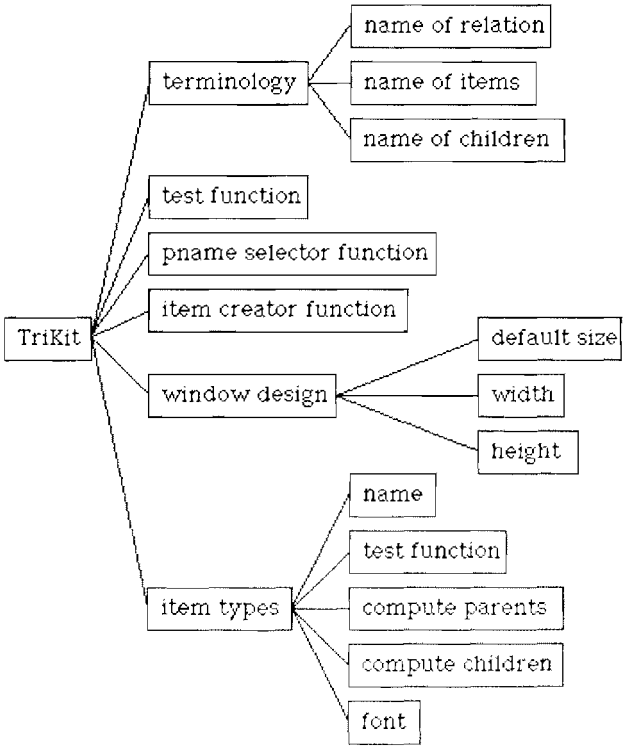


With WIDES the users are aware that code is being generated and they can learn the functions of the different pieces of code. This makes it easy to make enhancements on the implementation level. Users are incrementally writing code (creating a definition) by selecting high-level operations from a dynamic menu. For TRIKIT there is no need to show the code 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 27). The design environments 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 environments that generate the code. More explicit knowledge representations are required to make the design environments themselves extensible.

Figure 27. **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 trade-offs and design problems in this area of research are:

1. The usability barrier: It takes a major effort and large training costs to learn a complex system that offers extensive functionality.
2. 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.
3. 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 environments are:

1. They provide a powerful environment for rapid prototyping of a large class of systems.
2. They increase the control of the user over systems without requiring the user to learn many details.
3. 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).
4. Associated support tools make it easier to learn and work with complex systems.

Specific construction and design environments are used by different classes of users for a variety of different tasks. Our experience has shown that the use of design environments is not restricted to the inexperienced user: If the functionality offered by the design environment 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 environment 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. With design environments, learning processes are controlled and better restricted to that what is really needed.

It remains an open question whether we can succeed in considerably extending the functionality of the design environments 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, and gauges.

The domain of user interface design does not have a well-established terminology. Terms vary from system to system and from manufacturer to manufacturer. In addition, user interfaces are often designed or redesigned by nonspecialists. For these reasons, it has been difficult to find a vocabulary that enables the user to understand the descriptions for the required inputs. The meaning of the label Pname selector for items (Figure 24) is not obvious to someone who does not know this technical term of the menu system. To remedy shortcomings like this, we have to:

1. find a better conceptualization of the design task and use it to restructure the forms to make them easier to understand;
2. 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;
3. allow alternate modes of specification; explore more direct forms of manipulation of prototypes;
4. 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 environments 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 environment'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 environment 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 alternatively (for a further discussion of this problem, see Waters, 1986).

Other issues arising from the conceptual distance between a description and what it describes need to be further explored. With WIDES, 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).

We believe that human problem-domain communication is the most promising way of overcoming these problems. Construction kits with large numbers of generally useful building blocks provide a good basis for making computer more usable; but without the additional assistance of design environments, users are lost in the wealth of information and possibilities.

Acknowledgments. We thank our 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, including Helga Nieper-Lemke who developed TRISTAN, and Christian Rathke who developed FINANZ. Without their contributions, the described research effort would not have been possible.

Support. 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

Alexander, C. (1964). *The synthesis of form.* Harvard University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A pattern language: Towns, buildings, construction.* New York: Oxford University Press.

Bates, M., & Bobrow, R. J. (1984). Natural language interfaces: What's here, what's coming, and who needs it. In W. Reitman (Ed.), *Artificial intelligence applications for business* (pp. 179–194). Norwood, NJ: Ablex.

Boecker, H.-D., Fabian, F., Jr., & Lemke, A. C. (1985). WLisp: A window based programming environment for FranzLisp. *Proceedings of the First Pan Pacific Computer Conference,* 580–595. Melbourne, Australia: Australian Computer Society.

Boecker, H.-D., Fischer, G., & Nieper, H. (1986). The enhancement of under-

standing through visual representations. *Proceedings of the CHI '86 Conference on Human Factors in Computing Systems,* 44–50. New York: ACM.

Bolt, R. A. (1984). *The human interface.* Belmont, CA: Lifetime Learning Publications.

Buchanan, B. G., & Shortliffe, E. H. (1984). *Rule-based expert systems: The MYCIN experiments of the Stanford Heuristic Programming Project.* Reading, MA: Addison-Wesley.

Chambers, A. B., & Nagel, D. C. (1985). Pilots of the future: Human or computer? *Communications of the ACM, 28,* 1187–1199.

Draper, S. W. (1984). The nature of expertise in UNIX. *Proceedings of INTERACT '84, IFIP Conference on Human-Computer Interaction,* 182–186. Amsterdam: Elsevier Science Publishers.

Fabian, F. (1986). Fenster- und Menuesysteme in der MCK [Window and menu systems in human computer communication]. In G. Fischer & R. Gunzenhaeuser (Eds.), *Methoden und Werkzeuge zur Gestaltung benutzergerechter Computersysteme [Methods and tools for the design of user-oriented computer systems]* (pp. 101–119). Berlin & New York: Walter de Gruyter.

Fischer, G. (1983). Symbiotic, knowledge-based computer support systems. *Automatica, 19,* 627–637.

Fischer, G. (1987a). Cognitive view of reuse and redesign. *IEEE Software, Special Issue on Reusability, 4*(4), 60–72.

Fischer, G. (1987b). A critic for LISP. *Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy),* 177–184. Los Altos, CA: Kaufmann.

Fischer, G., & Lemke, A. C. (1988). Constrained design processes: Steps towards convivial computing. In R. Guindon (Ed.), *Cognitive science and its application for human-computer interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Fischer, G., Lemke, A. C., & Rathke, C. (1987). From design to redesign. *Proceedings of the 9th International Conference on Software Engineering,* 369–376. Washington, DC: IEEE Computer Society.

Fischer, G., Lemke, A. C., & Schwab, T. (1985). Knowledge-based help systems. *Proceedings of the CHI '85 Conference on Human Factors in Computing Systems,* 161–167. New York: ACM.

Fischer, G., & Schneider, M. (1984). Knowledge-based communication processes in software engineering. *Proceedings of the 7th International Conference on Software Engineering (Orlando, FL),* 358–368. Los Angeles, CA: IEEE Computer Society.

Hooper, K. (1986). Architectural design: An analogy. In D. A. Norman & S. W. Draper (Eds.), *User centered system design: new perspectives on human-computer interaction* (pp. 9–23). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1986). Direct manipulation interfaces. In D. A. Norman & S. W. Draper (Eds.), *User centered system design: new perspectives on human-computer interaction* (pp. 87–124). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Illich, I. (1973). *Tools for conviviality.* New York: Harper & Row.

Nieper, H. (1985). *TRISTAN: A generic display and editing system for hierarchical structures* (internal memo). Boulder, CO: Department of Computer Science, University of Colorado.

Olsen, D. R., Jr., Buxton, W., Ehrich, R., Kasik, D. J., Rhyne, J. R., & Sibert, J. (1984). A context for user interface management. *IEEE Computer Graphics and Applications, 4*(12), 33–42.

Rathke, C. (1986). *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache*

[ObjTalk: knowledge representation in an object-oriented language]. Unpublished doctoral dissertation, Fakultaet fuer Mathematik und Informatik, Universitaet Stuttgart, Federal Republic of Germany.

Riekert, W.-F. (1986). *Werkzeuge und Systeme zur Unterstuetzung des Erwerbs und der objektorientierten Modellierung von Wissen [Tools and systems supporting acquisition and object-oriented modeling of knowledge].* Unpublished doctoral dissertation, Fakultaet fuer Mathematik und Informatik, Universitaet Stuttgart, Federal Republic of Germany.

Robertson, G., McCracken, D., & Newell, A. (1981). The ZOG approach to man-machine communication. *International Journal of Man-Machine Studies, 14,* 461-488.

Simon, H. A. (1981). *The sciences of the artificial.* Cambridge, MA: MIT Press.

Waters, R. C. (1986). KBEmacs: Where's the AI? *AI Magazine, 7*(1), 47-56.

Wilensky, R., Arens, Y., & Chin, D. (1984). Talking to UNIX in English: An overview of UC. *Communications of the ACM, 27,* 574-593.

Williams, M. D., Tou, F. N., Fikes, R., Henderson, A., & Malone, T. W. (1982). RABBIT: Cognitive science in interface design. *Proceedings of the Cognitive Science Conference (Ann Arbor, Michigan),* 82-85. Cognitive Science Society.

Winograd, T. (1979). Beyond programming languages. *Communications of the ACM, 22,* 391-401.

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition: A new foundation for design.* Norwood, NJ: Ablex.

Woods, D. D. (1986). Cognitive technologies: The design of joint human-machine cognitive systems. *AI Magazine, 6*(4), 86-92.

HCI Editorial Record. First manuscript received May 19, 1986. Revisions received October 3, 1986, March 25, 1987, and August 7, 1987. Accepted by Ruven Brooks. Final manuscript received December 3, 1987. —*Editor*