University of Colorado at Boulder

Department of Computer Science

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

Constrained Design Processes: Steps Towards Convivial Computing

Gerhard Fischer and Andreas C. Lemke
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder, Colorado 80309

in Raymonde Guindon (ed.):
"Cognitive Science and its Application for Human-Computer Interaction"
Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp 1-58

Abstract

Our goal is to construct components of *convivial computer systems* which give people who use them the greatest opportunity to enrich their environments with the fruits of *their* vision. *Constrained design processes* are a means of resolving the conflict between the generality, power and rich functionality of modern computer systems, and the limited time and effort which casual and intermediate users want to spend to solve their problems without becoming computer experts.

Intelligent support systems are components which make it less difficult to learn and use complex computer systems. We have constructed a variety of *design kits* as instances of intelligent user support systems which allow users to carry out constrained design processes and give them control over their environment. Our experience in building and using these design kits will be described.

Acknowledgements

The research described here was supported by the University of Colorado, the Office of Naval Research (contract number: N00014-85-K-0842), Triumph Adler (Nuernberg, Germany), and the German Ministry for Research and Technology.

Constrained Design Processes: Steps Towards Convivial Computing

Gerhard Fischer and Andreas C. Lemke
Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder, Colorado 80309

Abstract

Our goal is to construct components of *convivial computer systems* which give people who use them the greatest opportunity to enrich their environments with the fruits of *their* vision. *Constrained design processes* are a means of resolving the conflict between the generality, power and rich functionality of modern computer systems, and the limited time and effort which casual and intermediate users want to spend to solve their problems without becoming computer experts.

Intelligent support systems are components which make it less difficult to learn and use complex computer systems. We have constructed a variety of *design kits* as instances of intelligent user support systems which allow users to carry out constrained design processes and give them control over their environment. Our experience in building and using these design kits will be described.

1. Introduction

Most computer users experience computer systems as unfriendly, uncooperative and requiring too much time and effort to get something done. Users find themselves dependent on specialists, they notice that *software is not soft* (i.e., the behavior of a system can not be changed without reprogramming it substantially), they have to relearn a system after they have not used it for some time, and they spend more time fighting the computer than solving their problem.

In this paper we will discuss what design kits can contribute to the goal of convivial computing systems. From a different perspective, design kits also contribute to two other major goals of our research: to construct intelligent support systems [Fischer 86] and to enhance incremental learning processes with knowledge-based systems [Fischer1987a]. In Section 2 we will briefly describe what we understand by convivial systems. One way of making (especially functionality-rich) systems more convivial is to provide intelligent support systems (Section 3). In Section 4 we argue that for certain classes of users and tasks there is a need for constrained design processes. In Section 5 we present methodologies and systems which support constrained design processes. In Section 6 we describe in detail some of the tools and the systems which we have built to support constrained design processes:

1. WLISPRC is a tool to customize WLISP (a window-based user-interface toolkit, based on LISP, developed by our research group over the last 6 years; [Fabian, Lemke 85; Boecker, Fabian, Lemke 85; Fabian 86]).
2. WIDES, a window design kit for WLISP, allows designers to build window-based systems at a high level of abstraction and it generates the programs for this application in the background.
3. TRIKIT is a design kit for TRISTAN. TRISTAN [Nieper 85] is a generic tool for generating graphi-

cal representations for general graph structures. TRIKIT uses a form-based approach to allow the designer to combine application specific semantics with the generic tool.

In Section 7 we briefly describe our experience with using these systems. Section 8 relates our systems to other work in this area, and the last section discusses a few conclusions drawn from this work.

2. Convivial Computer Systems

Illich [Illich 73] has introduced the notion of "convivial tools" which he defines as follows:

> *Tools are intrinsic to social relationships. An individual relates himself in action to his society through the use of tools which he actively masters, or by which he is passively acted upon. To the degree that he masters his tools, he can invest the world with his meaning; to the degree that he is mastered by his tools, the shape of the tool determines his own self-image. Convivial tools are those which give each person who uses them the greatest opportunity to enrich the environment with the fruits of his or her vision.*
>
> *Tools foster conviviality to the extent to which they can be easily used, by anybody, as often or as seldom as desired, for the accomplishment of a purpose chosen by the user.*

Illich's thinking is very broad and he tries to show alternatives for future technology-based developments and their integration into society. We have applied his thoughts to information processing technologies and systems [Fischer 81] and believe that conviviality is a dimension which sets computers apart from other communication technologies. All other communication and information technologies (e.g., television, videodiscs, interactive videotex) are passive, i.e. users have little influence to shape them to their own taste and their own tasks. They have some selective power but there is no way that they can extend system capabilities in ways which the designer of those systems did not directly foresee.

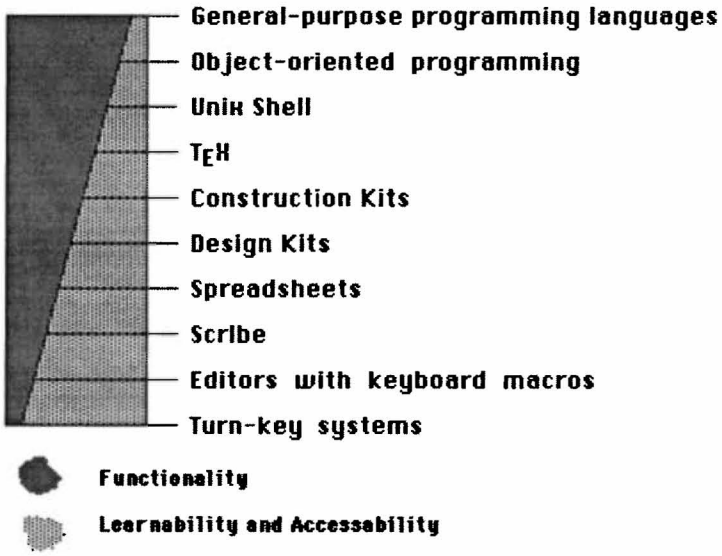

Figure 2-1: The spectrum of conviviality

We do not claim that currently existing computer systems are convivial. Most systems belong to one of the extremes of the spectrum of conviviality (Figure 2-1):

1. **General purpose programming languages:** They are powerful, but they are hard to learn, they are often too far away from the conceptual structure of the problem, and it takes too long to get a task done or a problem solved. This class of systems can be adequately described by the Turing tar-pit (defined by Alan Perlis; see [Hutchins, Hollan, Norman 86]):

   > *Beware the Turing tar-pit, in which everything is possible but nothing of interest is easy.*

2. **Turn-key systems:** They are easy to use, no special training is required, but they can not be modified by the user. This class of systems can be adequately described by the converse of the Turing tar-pit:

Beware the over-specialized system where operations are easy, but little of interest is possible.

Starting from both ends, there are promising ways to make systems more convivial. Coming from the "general purpose programming languages" end of the spectrum, object-oriented programming (in SmallTalk [Goldberg 81] or ObjTalk [Rathke 86]), user interface management systems, programming environments and command languages like the UNIX shell are efforts to make systems more accessible and usable. Coming from the other end, good turn-key systems contain features which make them modifiable by the user without having to change the internal structures. Editors allow users to define their own keys ("key-board macros") and modern user interfaces allow users to create and manipulate windows, menus, icons etc., at an easy to learn level.

Turn-key systems appear as a monolithic block (Figure 2-2). The user can choose to use them if their functionality is appropriate. But they become obsolete if they cannot meet a specific requirement.

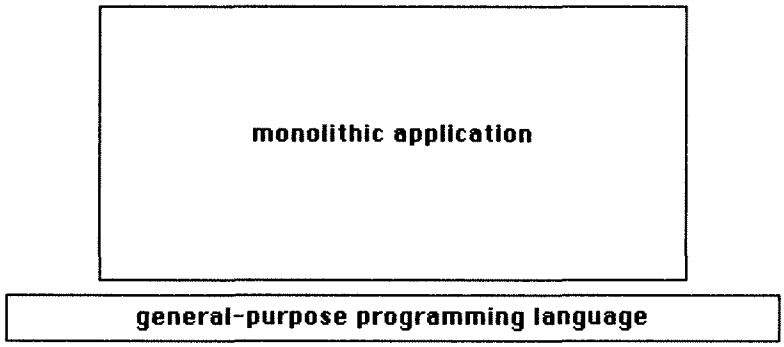

Figure 2-2: Turn-key systems

The user should have control over a tool on multiple levels. Figure 2-3 shows the levels of control for the EMACS editor [Stallman 81; Gosling 82]. The keystroke level together with its special purpose extensions (lisp mode, etc.) is most frequently and most easily used. The lower levels gradually provide more functionality but require more knowledge about the implementation of the editor. Due to this structure, EMACS is perceived as a convivial tool that can be extended and adapted to many different needs.

Despite our goal of making computer systems more convivial, i.e., giving more control to the user, we *do not believe that more control is always better.* Many general advances in our society (e.g., automatic transmission in automobiles) and those specifically in computing are due to the automation of tasks which before had to be done by hand. Assemblers freed us from keeping track of memory management, high level languages and compilers eliminated the need to take specific hardware architectures into account, and document production systems allow us to put our emphasis on content instead of form of written documents.

The last domain illustrates that the right amount of user control is not a fixed constant, but depends on the users and their tasks. Truely convivial tools should give the user any desired control, but they should not require that it be exercised. In this sense, the text formatting systems T_EX and Scribe [Furuta, Scofield, Shaw 82], show the following differences (see also Figure 2-1):

1. The T_EX user is viewed as being an author who wants to position objects exactly on the

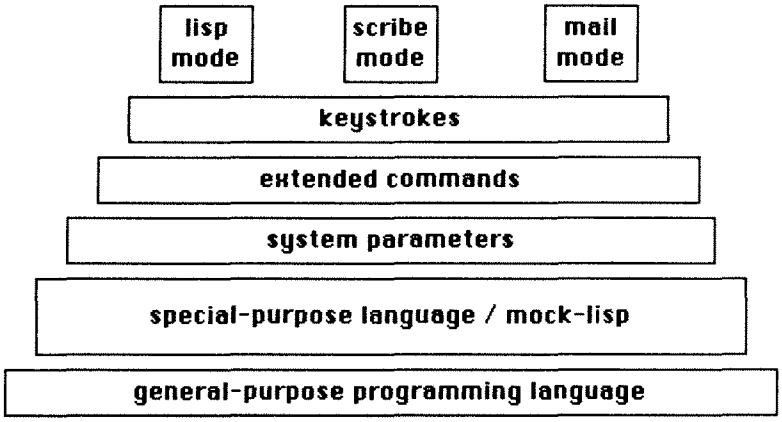

Figure 2-3: Levels of control over the EMACS editor

printed page, producing a document with the finest possible appearance. The user has to exercise a rather large amount of control. The emphasis is on power and expressiveness of the formatting language.

2. The Scribe user is viewed as an author who is more interested in easily specifying the abstract objects within the document, leaving the details of the appearance of objects to an expert who establishes definitions that map the author's objects to the printed page. The Scribe user usually exercises little control. Although Scribe offers substantial control over the appearance of a document, it does not allow to specify everything that is possible with T_EX. Scribe's emphasis is on the simplicity of its input language and on support by writer's workbench tools.

The development of convivial tools will break down an old distinction: *there will be no sharp border line between programming and using programs* -- a distinction which has been a major obstacle for the usefulness of computers. Convivial tools will remove from the "meta-designers" (i.e., the persons who design design-tools for other people) the impossible task of anticipating all possible uses of a tool and all people's needs. Convivial tools encourage users to be actively engaged and to generate creative extensions to the artifacts given to them. Their use and availability should not be restricted to a few highly educated people. Convivial tools require to replace "Human Computer Communication" by "Human Problem-Domain Communication" . Human Problem-Domain Communication is an important step forward, because users can operate within the semantics of their domain of expertise and the formal descriptions closely match the structures of the problem domain.

Convivial tools raise a number of interesting questions, which we will investigate in our future research: Should systems be *adaptive* (i.e., the system itself changes its behavior based on a model of the user and the task [Fischer, Lemke, Schwab 85]) or should systems *be adaptable by the user* ? Should systems be composed of simple or intelligent tools [Norman 86]? *Simple tools* can have problems, because they require too much skill, time and effort from the user. It is, for example, far from easy to construct an interesting model using a sophisticated technical construction kit [Fischer, Boecker 83]. *Intelligent tools* can have problems because many of them fail to give any indication of how they operate and what they are doing; the user feels like an observer, watching while unexplained operations take place. This mode of operation results in a lack of control over events and does not achieve any conviviality.

3. Intelligent Support 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 3-1 we give an indication of the complexity (in number of objects, tools and amount of written documentation) of modern computer systems.

Number of Computational Objects in Systems

EMACS:

- 170 function keys and 462 commands

UNIX:

- more than 700 commands and a large number of embedded systems

LISP-Systems:

- FRANZ-LISP: 685 functions
- WLISP: 2590 LISP functions and 200 ObjTalk classes
- SYMBOLICS LISP MACHINES: 19000 functions and 2300 flavors

Amount of Written Documentation

Symbolics LISP Machines:

- 10 Books with 3000 Pages
- does not include any application programs

SUN workstations:

- 15 books with 4600 pages
- additional Beginner's Guides: 8 books totaling 800 pages

Figure 3-1: Quantitative Analysis of Some Systems

In our research work we have used the computational power of modern computer systems to construct a

variety of *intelligent support systems* (see Figure 3-2). These support systems are called *intelligent*, because they have knowledge about the task, knowledge about the user and they support communication capabilities which allow the user to interact with them in a more "natural" way. Some of the components (e.g., for explanation and visualization) are specifically constructed to overcome some of the negative aspects of intelligent tools as mentioned above (e.g., the user should not be limited to an observer, but should be able to understand what is going on).

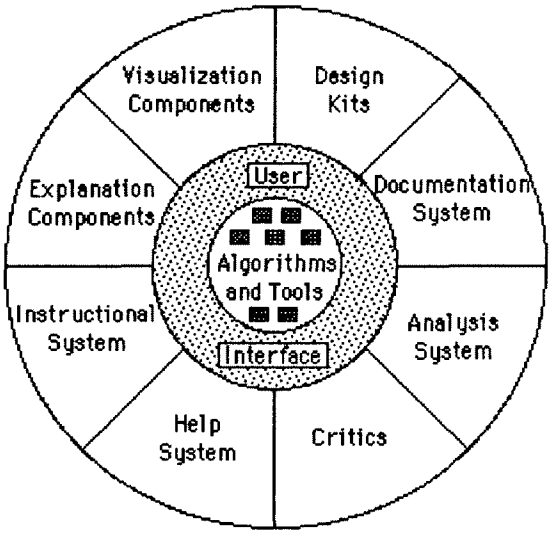

Figure 3-2: The architecture of intelligent support systems

By constructing these intelligent support systems we hope to increase the conviviality of systems. Some of our prototypical developments are described in the following papers:

- documentation systems in [Fischer, Schneider 84],
- help systems in [Fischer, Lemke, Schwab 85],
- critics in [Fischer1987a],
- visualization tools in [Boecker, Fischer, Nieper 86] and
- design kits in section 6 of this chapter.

4. The Need for Constrained Design Processes

Alan Kay [Kay 84] considers the computer as the first *metamedium* with degrees of freedom for representation and expression never before encountered and as yet barely investigated. This large design space makes design processes very difficult. Much experience and knowledge is needed if this space is to be successfully used. Especially for those who use the computer only as a tool, this space is overwhelming and can choke any attempts at making the computer convivial. Constraining the design space in a user and domain dependent way can make more design processes tractable, even for non computer experts.

With our research goals in mind, we encounter a difficulty in terminology: our users should not just be consumers but also designers. Therefore, we have to introduce the notion of meta-designer for the group of people who build design tools for other people. For simplicity, we will use the pair "designer and user" instead of "meta-designer and designer". Having pointed out this distinction, a "user" is still not a clearly defined concept. In some cases he/she may be the domain expert (i.e., a person who knows little about computers but much about a certain application domain), in other cases he/she may be a system designer who uses a knowledge representation formalism or a user interface toolkit. Most of the design kits described in section 6 build a bridge between these two levels.

The following objectives generate a need for constrained design processes:

1. *to enhance incremental learning of complex systems* and to delimit useful microworlds -- inexperienced users should be able to get started and do useful work when they know only a small part of the system [Fischer1987a];
2. *to increase subjective computability* (e.g., by eliminating prerequisite knowledge and skills and by raising the level of abstraction);
3. *to make experts more efficient* (e.g., they can rely on tested building blocks and they do not have to worry about details);
4. *to guide users in the relevant context* so they can choose the next steps (e.g., WIDES (see Section 6.2) provides in the code window the corresponding ObjTalk definition of what users would have to do if the WIDES were not available);
5. *to lead the user from "chaos to order"* (e.g., the primitives of a programming language or the basic elements of a technical construction kit give little guidance on how to construct a complex artifact to achieve a certain purpose).

4.1 User Modifiability and User Control

Why is there a need for user modifiability and user control? The specification process of what a program should do is more complex and evolutionary than previously believed. This is especially true for ill-structured problems like those which arise in areas like Artificial Intelligence and Human-Computer Communication. Computer systems in these areas are *open systems*, their requirements cannot be defined in detail at program writing time but will arise dynamically at program run time. Programs have to cope with "action at a distance." Many non-expert users who used the computer mostly to support them in carrying out routine cognitive skills like text processing are now more and more requiring individualized support for increasingly demanding cognitive skills like information retrieval, visualization support in understanding complex systems, explanations, help and instruction.

The goal of making tools modifiable by the user does not imply transferring the responsibility of good tool

design to the user. It is probably safe to assume that normal users will never build tools of the quality a professional designer would. But this is not the goal of convivial systems. Only if the tool does not satisfy the needs and the taste of the users (which they know best themselves) then should they carry out a constrained design process to adapt it. The strongest test of a system with respect to user modifiability and user control is not how well its features conform to anticipated needs but how well it performs when one wants to do something the designer did not specifically foresee although it is in the system's global domain of application.

Pre-designed systems are too encapsulated for problems whose nature and specifications change and evolve. A useful system must accommodate these changing needs. The user must have some amount of control over the system. Suppose we design an expert system to help lawyers. As a lawyer uses this system, the system should be able to adjust to his or her particular needs which cannot be foreseen because they may be almost unique among the user population. Furthermore, in many cases this adjustment cannot be done by sending in a request for modification to the original system developers, because they may have problems understanding the nature of the request. The users should be able to make the required modifications in the system themselves.

4.2 Support for the Casual and Intermediate Users

The rich functionality of modern computer systems made two classes of users predominant: *casual* and *intermediate* users. The demands made on users memory and learning ability are illustrated by a quantitative analysis of the systems used in our research (Figure 3-1).

Even if users are experts in *some* systems, there will be many more systems available to them where they are at best casual users. *Casual users* need in many cases more control and more variability than turn-key systems are able to offer, but at the same time it should not be necessary to know a system completely before anything can be done. Another issue is also crucial for casual users: the time to relearn a system after not having used it for some time. It seems a safe assumption that a system which was easy to learn the first time should not be too difficult to relearn at a later time.

In order to successfully exploit the capabilities of most complex computer systems, users must have reached an *intermediate* level of skills and knowledge. Incremental learning processes [Fischer1987a], which extend over months and years, are required to make the transition from a novice to an expert. One intrinsic conflict in designing systems is caused by the demand that these systems should have *no threshold and no ceiling.* There should be entry points which make it easy to get started and the limitations of the systems should not be reached soon after. Constrained design processes are one way to partially resolve this design conflict. Our window design kit (see Section 6.2) has exactly this goal: to serve as an entry point to the full generality of our user interface construction kit. The code for various types of windows can be created automatically by making selections from a suggestion menu. If more complex windows are desired, one can modify the generated code. Under this perspective it serves as a *transient object*: if someone knows the underlying formalisms well enough, then there is no need any more to use the design kit.

4.3 Support for Rapid Prototyping

Constrained design processes are *not* only useful to produce transient objects. Experts can also take advantage of them if the functionality required is within the scope of the constrained design processes. The advantages of restricting ourselves to the limits of constrained design processes are: the human effort is smaller (e.g., less code to write), the process is less error-prone (because we can rely on tested building blocks) and users have to know less to succeed (e.g., no worries about low-level details). These advantages are especially important for a rapid prototyping methodology where several experimental systems have to be constructed quickly.

4.4 From Design to Redesign

We argued before that complex systems can never be completely predesigned, because their applications cannot be precisely foreseen, and requirements are often modified as the design and the implementation proceeds. Therefore, real systems must be continuously *redesigned* [Fischer, Kintsch 86]. *Reuse* of existing components is an important part of this process. 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 functioning of these parts must be understood. An important question concerns the level of understanding that is 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 knowledge representation language ObjTalk) and tools are one step in the direction of making it easier to modify an existing system than to create a new one.

5. Methodologies and Systems to Support Constrained Design Processes

Informal experiments [Fischer1987a] indicate that the following problems prevent users from successfully exploiting the potential of high functionality systems:

- users do not know about the existence of tools,
- users do not know how to access tools,
- users do not know when to use these tools,
- users do not understand the results which are produced by the tools,
- users cannot combine, adapt and modify a tool to their specific needs.

In this section we describe how constrained design processes, based on different methodologies and supported by different kinds of systems, can overcome some of these problems.

5.1 Selection

Selection of tools from a set of tools seems to be the least demanding method to carry out a constrained design process. But in realistic situations, it is far from being trivial. Different from a Swiss army knife, which has at most 15 different tools (Figure 5-1), we may have hundreds or thousands (Figure 3-1) of tools in a high functionality computer system.

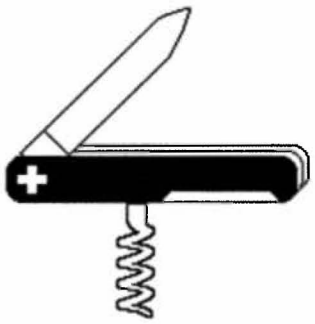

Figure 5-1: Selection of tools: the Swiss army knife

The CATALOG, a tool which we recently built to access the many tools and application systems in our WLISP system, simplifies the selection process. The iconic representations help users to see what is there and it provides clues about systems they might be interested in.

Selection systems can be made more versatile by giving the user the possibility to make adjustments or set parameters (like specifying options to commands). They require that most work is done by the designer in anticipation of the needs of the user. This strategy leads to systems containing a large number of tools many of which may never be used and which are, consequently, unnecessarily complex.

Are selection type systems all that we need? We do not think so and agree with Alan Kay [Kay 84] who notes:

> *Does this mean that what might be called a driver-education to computer literacy is all most people will ever need - that one need only learn how to "drive" applications programs and need never learn to program? Certainly not. Users must be able to tailor a system to their wants. Anything less would be as absurd as requiring essays to be formed out of paragraphs that have already been written.*

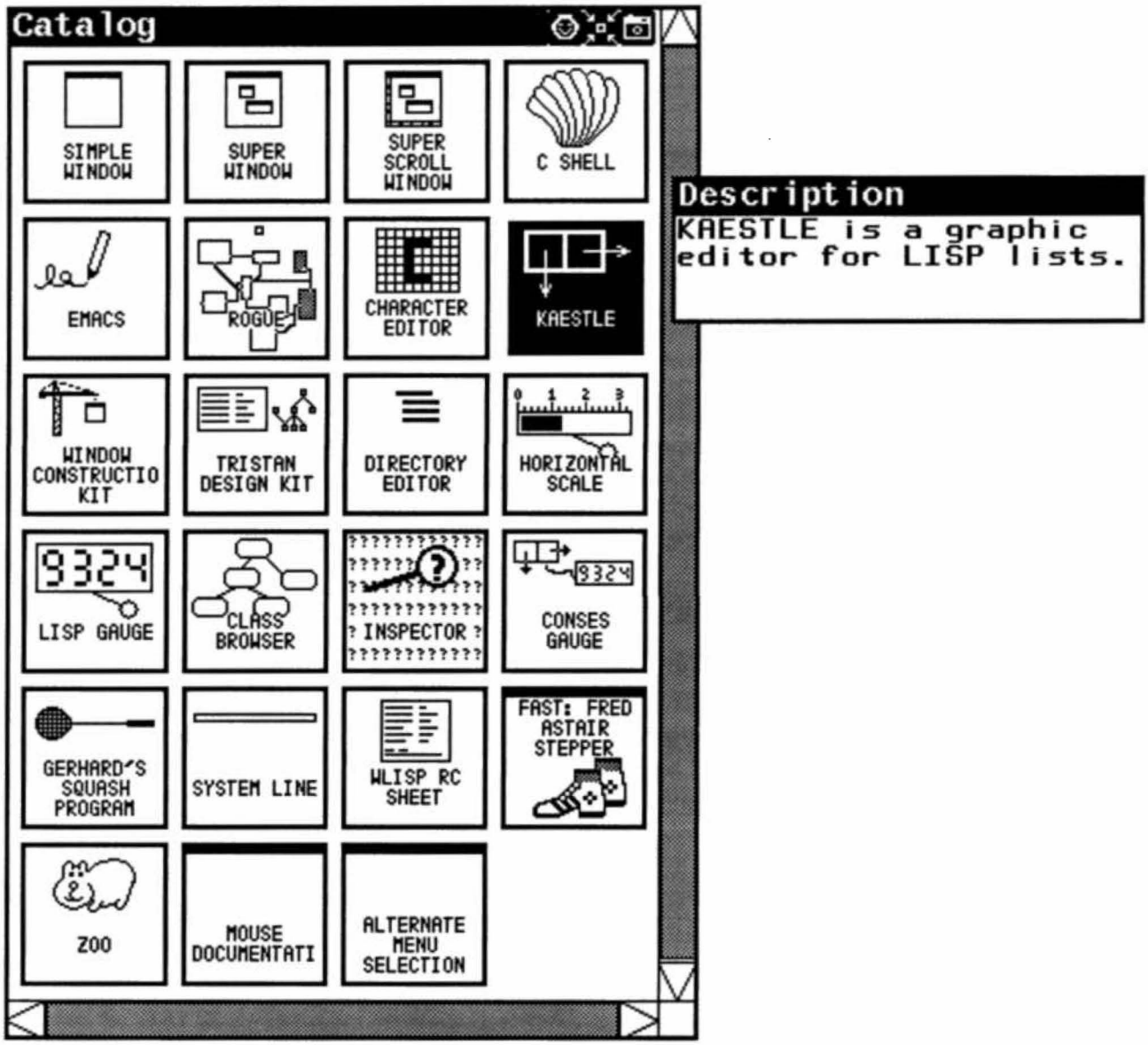

Figure 5-2: The CATALOG: a tool to simplify selection processes

5.2 Simple Combination

Believing in recursive function theory, we know that we can compute anything with a set of very simple functions and powerful ways of combination like function definition and recursion. But the more interesting combinations are too complicated and require too many intermediate levels to get to the level of abstraction the user can operate at.

A good example of a simple combination process is illustrated with the electric drill in Figure 5-3 (the basic design of TRIKIT in Section 6.3 is very similar).

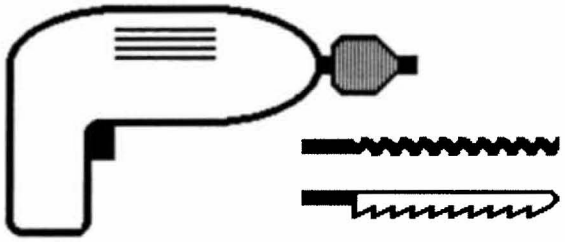

Figure 5-3: Simple combination of tools: the electric drill

Other simple combination processes allow the user to define keyboard macros in extensible editors. The combination method in this case is just a simple sequencing operation. Another example is the concept of a pipe in UNIX which allows to use the output of one tool as the input to another tool. Direct manipulation styles of human-computer interaction are also based on a simple combination process: any output which appears on the screen can be used as an input (a concept called "interreferential I/O" by Draper in [Norman, Draper 86]).

Combination processes become more difficult if there are many building blocks to work with, if the number of links necessary to build a connection increases and if compatibility between parts is not obvious.

5.3 Instantiation

Instantiation is another methodology to carry out constrained design processes. The adjustable wrench (Figure 5-4) can be thought of (in the terminology of object-oriented programming) as being the class of all wrenches which is instantiated through an adjustment process to fit a specific bolt. A restricted form of programming in an object-oriented formalism like ObjTalk can be done by creating instances of existing classes. Classes provide a set of abstract descriptions and if enough classes exist, we can generate a broad range of behavior. In KBEmacs [Waters 85], instantiation is used to turn abstract cliches into program code.

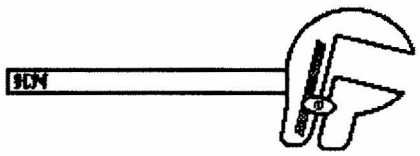

Figure 5-4: Instantiation: the adjustable end wrench

5.4 Design Kits

Computer-supported design kits, as we try to envision and construct them, should not be restricted to providing the building blocks for a design, but they should support the process of composing interesting systems within the application domain. The building blocks should have self-knowledge and they should be more like active agents than like passive objects.

Design kits can be differentiated from:

1. *construction kits:* the elements of construction kits (e.g., the mixins and the general classes

in WLISP (Figure 6-1 and 6-2); the parts in a technical construction system) are not particularly interesting by themselves but serve as building blocks for larger structures. Examples of excellent construction kits can be found in the software of Electronic Arts (e.g., the PinBall and the Music construction kits), in technical areas (e.g., FischerTechnik [Fischer, Boecker 83]) and in the toy world (e.g., LEGO).

2. *tool kits:* tool kits provide tools which serve specific purposes (e.g., the more specific classes in WLISP (Figure 6-1) or the entries in the catalog (Figure 5-2)); but a tool kit itself provides no guidance how to exploit the power of it to achieve a certain task. Contrary to components of construction kits, tools do not become part of the system constructed.

Design kits are *intelligent support systems* (Figure 3-2), which we see as integral parts of future computer systems. We believe that each system that allows user modifiability should have an associated design kit. Design kits can contribute towards the achievement of the following goals: to at least partially resolve the basic design conflict between generality and power versus ease of use; to make the computer behind a system invisible; users should be able to deal primarily with the abstractions of the problem domain (human problem-domain communication); and, finally, to protect the user from error messages and attempting illegal operations.

Design kits provide prototypical solutions and examples which can be modified and extended to achieve a new goal instead of starting from scratch; they support a "copy&edit" methodology for constructing systems through reuse and redesign of exisiting components [Fischer, Lemke, Rathke 87].

5.5 Object-Oriented Programming

Objects encapsulate procedures and data and are the basic building blocks of object-oriented programming. Objects are grouped into classes and classes are combined in an inheritance hierarchy (Figure 6-2). This inheritance hierarchy supports differential description (object y is like object x *except* u,v,...). Object-oriented formalisms [Lemke 85; Rathke 86] support constrained design processes through instantiation of existing classes (see Section 5.3) and through the creation of subclasses which can inherit large amounts of information from their superclasses. Many tasks can be achieved before one has to use the full generality of the formalism by defining new classes. New programming methodologies like differential programming and programming by specialization are supported.

6. Examples of Design Kits

The design kits described in this section have been implemented within the WLISP environment [Fabian, Lemke 85] (Figure 6-1 shows an example screen). WLISP is an object-oriented user interface toolkit and programming environment implemented in FranzLisp and ObjTalk [Rathke 86], an object-oriented extension to Lisp.

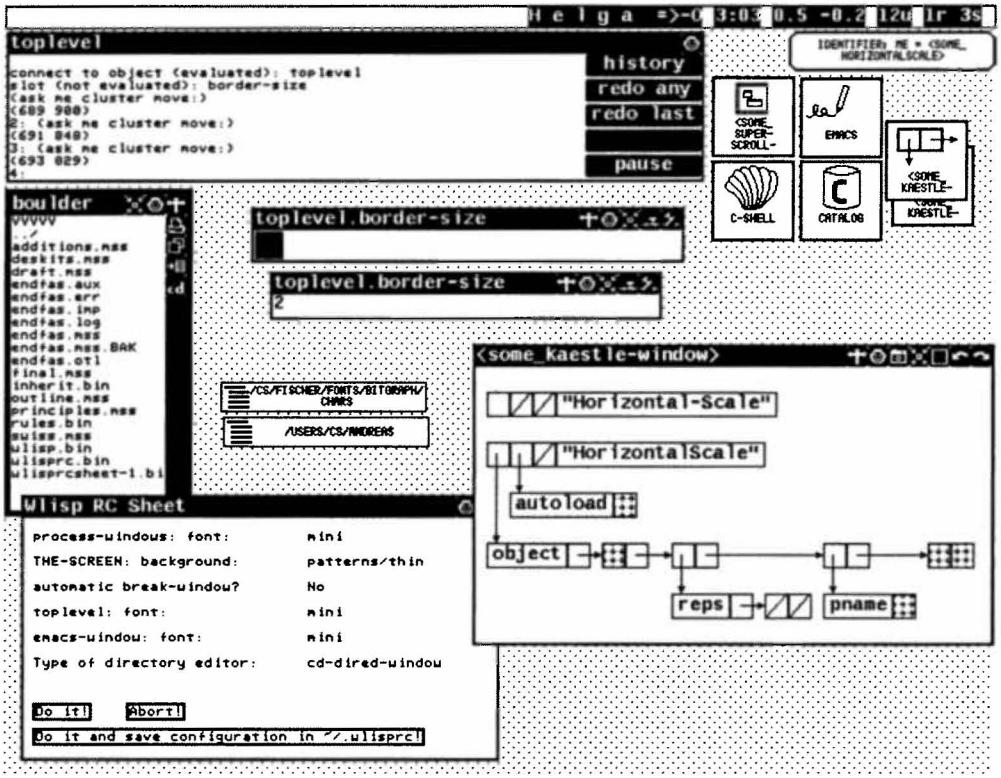

Figure 6-1: The WLISP programming environment

The object-oriented nature of the system provides a good framework to represent the entities of the toolkit (windows, menus, push buttons, etc.). Figure 6-2 shows an ObjTalk inheritance hierarchy of some simple window classes. In the following sections we will see how design kits can help to build new applications from the building blocks of this hierarchy.

6.1 WLISPRC[1]

Characterization of the Problem Situation: Like many large systems, WLISP has several configuration parameters that make it adaptable to various uses:

- A *programmer* may want to have a debugger immediately available and see system resource statistics,

[1]The name WLISPRC has been chosen because it creates a system initialization file for WLisp. In Unix jargon the names of these files are commonly composed of the system name and the letters 'rc'.

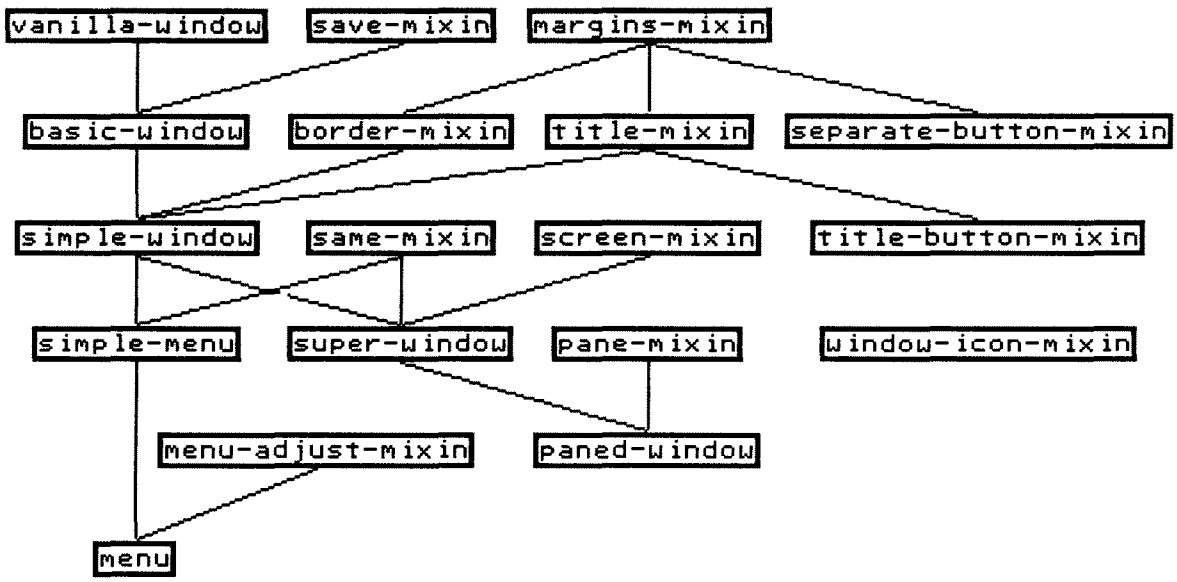

Figure 6-2: The inheritance hierarchy of windows

- whereas if the system is used for text processing, a text editor, a directory editor and a text formatter should be in close reach.

Many of these parameters are not easy to access; their names are hard to remember; the user may not even know of their existence. A second problem is that most of them live in the dynamic environment of the executing system only and are reset to their default values when the system is rebooted. For this reason, a mechanism must be provided to save these parameters for future sessions.

Approach: A system configuration sheet has been built which shows certain system parameters. It allows their values to be edited in a constrained way and to permanently store the state of the system (Figure 6-3).

.FIGURE /cs/fischer/lib/bgscreens/wlisprc.bin 7.6cm -m 2

Figure 6-3: The WLISPRC sheet

6.1.1 Description of WLISPRC

We have considered two basic approaches to customize the WLISP programming environment:

1. Some systems have a configuration file, possibly with a specific editor which helps to fill in correct parameters, and
2. the user modifies the state of the system while using it through means provided inside the system (e.g., manipulation of the screen display with the mouse), and the system stores those settings immediately or at the end of the session in permanent memory (disk file).

WLISPRC is a system configuration sheet with two functions:

1. Setting certain system parameters (e.g., fonts of certain windows) which are shown in the

sheet and which are otherwise only accessible through the LISP interpreter. At the same time, it makes sure that only legal values are entered (constrained editing). This is done using menu selection and choice fields which circle through a set of values (Yes/No toggles are a special type of them).

2. Saving the system configuration for later use.

In addition to the parameters shown in the sheet, the configuration includes the current size and location of system windows on the screen and some information pertaining to currently loaded applications.

```
;; The font of the toplevel window. Ask is the form to send a message to an
;; object (here the toplevel window).
(ask toplevel font = mini)

;; Size and position of the toplevel window.
(ask toplevel screenregion = (1 884 542 120))

;; Font default for process windows is mini.
(setq process-window-font 'mini)

(ask THE-SCREEN background = patterns/thin)

;; Do not pop up the break window automatically.
(setq automatic-break-window 'nil)

(ask emacs-window repslot: font (default process-window-font))

(setq directory-window cd-dired-window)

;; Load the sysline window and set its size, position, and mode.
(load-if-needed 'sysline window:dir)
(ask sysline-window-1 set:
     (screenregion = (0 1006 768 18))
     (flags = ""))

;; Load the catalog.
(load-if-needed 'catalog/system-catalog window:dir)
(ask Catalog move: 1 561)
(ask Catalog size = 283 283)

;; Load the directory editor and create an instance.
(load-if-needed 'directory-editor window:dir)
(setq current-directory (path:pwd))
(ask ,(ask ,directory-window instantiate:
           (screenregion = (616 534 146 260)))
     totop: nil)

;; Load the window identifier.
(load-if-needed 'identify window:dir)
(ask identifier move: 608 972)
```

Figure 6-4: The system initialization file generated by WLISPRC

Figure 6-4 shows a system initialization file as generated by WLISPRC. Some of its entries have been commented for illustration purposes. The file contains Lisp and ObjTalk code.

6.1.2 Evaluation

Although system initialization files which can contain arbitrary program code provide the same, even a basically unbounded functionality, WLISPRC gives to most users much more *subjective control* over screen layout and many other parameters. Note that WLISPRC does not exclude the use of a regular initialization file.

WLISPRC is an example of the *reduce learning principle* . It can be easily seen that considerable knowledge would be necessary to write this file directly. The user would have to know the names of fonts, the names of the objects, and the slots where the parameters are to be stored.

In the case of the size and position of a window (`screenregion`), the numeric coordinates do not say much about the overall appearance on the screen. If users see the real screen layout then they can tell immediately whether it is satisfying.

Instead of using this low level language, WLISPRC lets users make these specifications by directly moving windows to desired places, selecting fonts from menus etc.. The mode of interaction can be characterized as selection and direct editing and manipulation.

There are some problems which are not sufficiently addressed yet:

- It is not easy to know what the system's idea of its state is. The sheet shows only those parameters that can be modified using the sheet. Layout parameters which are affected through direct manipulation with the mouse are not reflected. What is a good way to do that?
- The meaning of some parameters (e.g., `automatic break window`) may be obscure to beginners because they may not even have encountered a situation where they are relevant. A user model could help to control whether this parameter should be listed and, if yes, in which way.
- It is a priori not clear which parameters of a system are part of its state and which of them are only relevant for the current context. In that WLISPRC has a set of permanent parameters that it stores in a disc file, it provides WLisp with some knowledge about this problem. For a parameter like the font of the toplevel window, however, this decision cannot be made generally. The reason for choosing a particular small font may be to be temporarily able to see a complete listing of some particular large data, but it also may be that the original font doesn't have a certain character that frequently occurs in outputs to this window.

  The classification of parameters may also change with the uses of the system and when it is being modified. It may be necessary to extend the system's notion of its state (e.g., "The following files should be loaded automatically"). How can the system be told to save more state? Currently there is only a programming language level interface for this purpose.
- Why not simply save everything and avoid worrying about the systems state? If the system is in some erroneous state or contains a lot of "garbage", it is certainly necessary to either suppress automatic state saving or to be able go back to an earlier version.

6.2 WIDES: A Window Design Kit

Characterization of the Problem Situation: Window systems and user interface tool kits have a rich functionality. There are text windows and graphic windows that may have controls like menus and push buttons associated with them. There are various text, graphic, and network editors that may be adapted for particular applications.

Currently, the use of these components requires a considerable expertise which may only be acquired through an extended learning and experimentation period. The goal of WIDES is

1. to reduce the knowledge required to use the components,
2. to support this learning process and
3. to provide guidelines to structure a toolkit in such a way that useful work can be done when only a small part of it is known.

Approach: Building design kits is a way to address these goals. WIDES provides a safe learning environment in which no fatal errors are possible and in which enough information is provided in each situation to make sure that there is always a way to proceed. The design kit allows users to create simple window types for their applications.

6.2.1 Description of WIDES

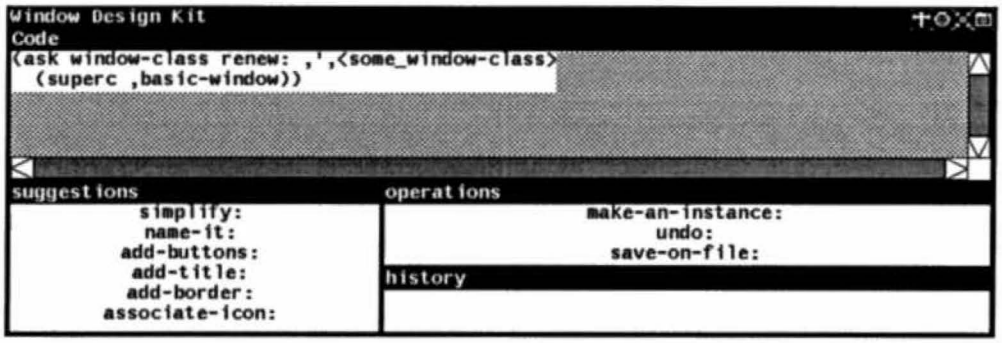

Figure 6-5: Initial state of WIDES

The initial state of the system is shown in Figure 6-5. 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*.

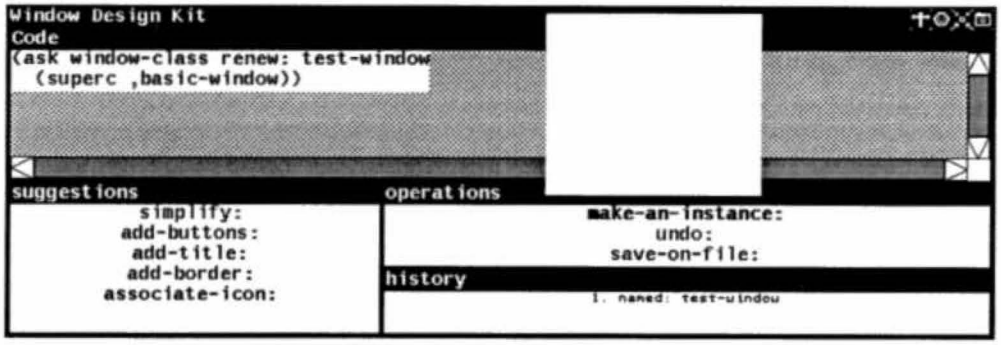

Figure 6-6: An instance of the current window definition has been created.

Selection of the "`name-it:`" entry of the suggestions menu makes the system ask for a name the user wants to call the window type to be built. Selection of the "`make-an-instance:`" item of the operations menu creates a window (an *instance* of the type) that corresponds to the current definition in the code pane. This definition describes a very basic type of window (Figure 6-6); there is no border, no title bar yet; just a rectangular white area. Nevertheless, this window has a set of properties which are inherited from its superclass `basic-window`. It reacts on mouse clicks by showing the `window-menu`, a menu with operations like `move`, `reshape` etc..

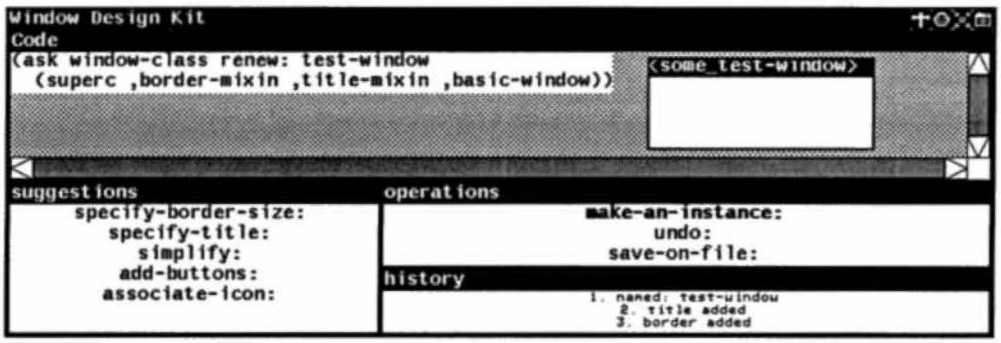

Figure 6-7: Title and border have been added to the window type.

Selection of "`add-title:`" and "`add-border:`" produces the state of Figure 6-7. Two superclasses (`border-mixin`, `title-mixin`) have been added to the definition and 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 to a tree like regime. Once a key decision (like having a title) has been made, then its menu item is replaced by suggestions for more detailed descriptions. This provides the user with some guidance about reasonable next steps, eliminates illegal operations, and reduces the information overload.

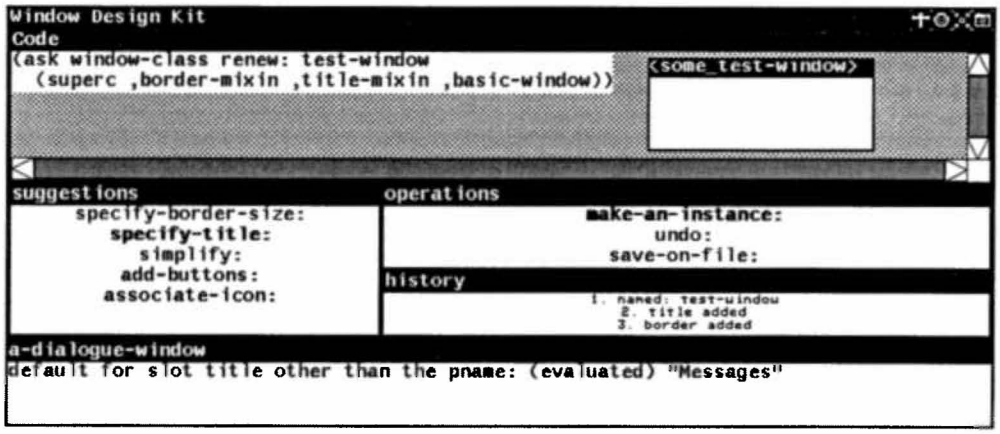

Figure 6-8: Specification of the title

The next figure (Figure 6-8) 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 6-9 shows that the input has been added as a default for the title slot.

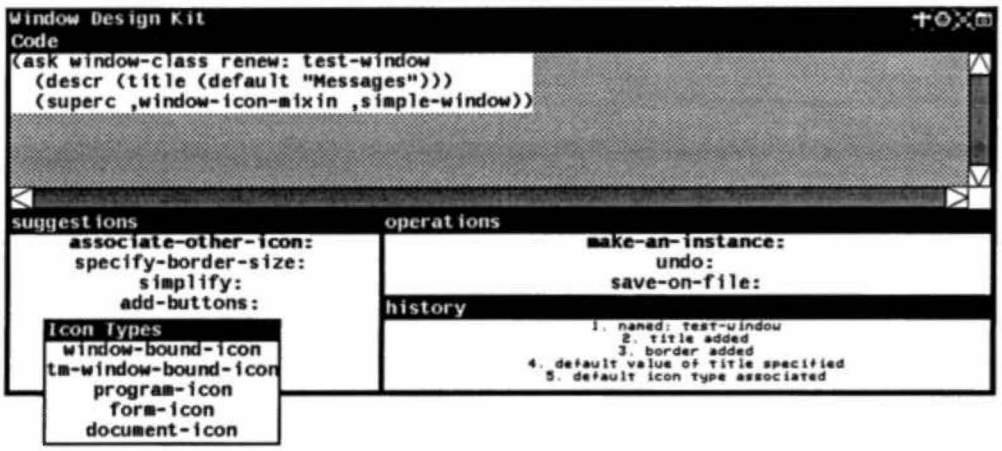

Figure 6-9: Association of a different icon to the window

If specific inputs are required, we cannot expect the user to know what the legal inputs are. Therefore, as in Figure 6-9 where the user associates the window with an icon, a menu of alternatives is displayed (see the pop up menu at the bottom). Figure 6-10 shows a window and an icon of the selected type.

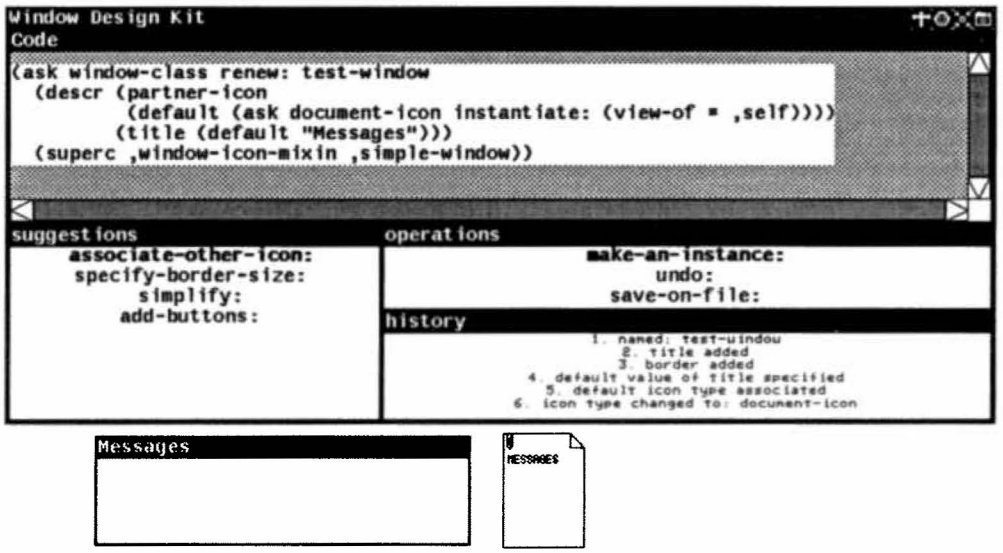

Figure 6-10: The window and its associated icon

An even more complex modification is demonstrated in Figure 6-11. Windows can be associated with push buttons such as those in the upper right corner of the window design kit window. Clicking the button with the mouse causes a message to be sent to the window. As an extension of the push buttons in the

title bar supplied by default (the two right-most ones), a button for shrinking the window to its associated icon is to be added. After selecting "`add-more-buttons-to-title-bar:`" from the suggestions menu, the user is asked to choose a button icon and a message from two menus. The shrink button appears as the leftmost button in the instance of `test-window` in Figure 6-11.

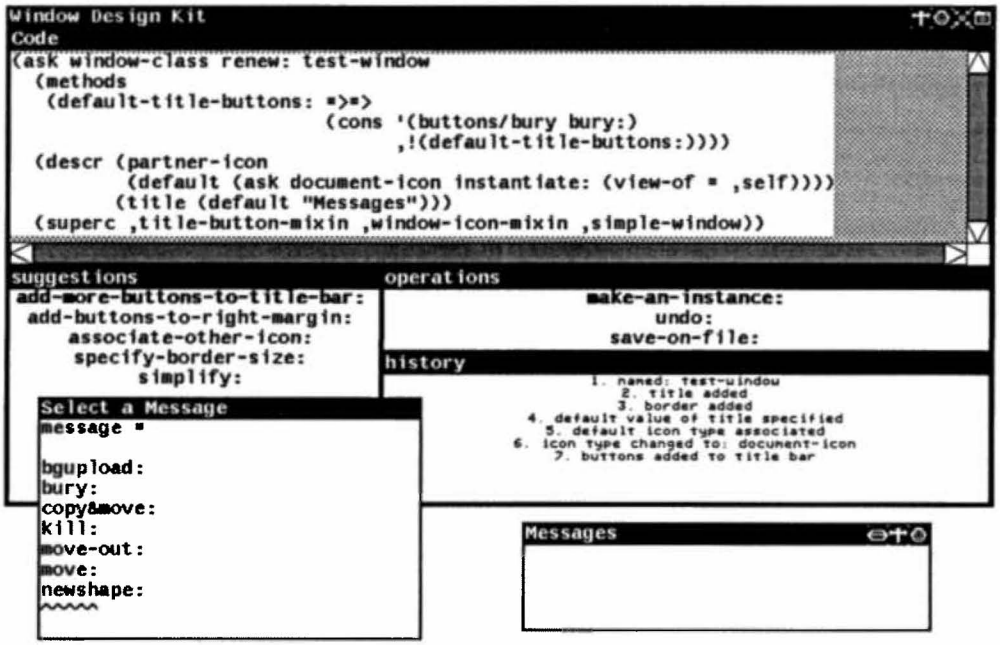

Figure 6-11: Adding a button to the title bar

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

6.2.2 Evaluation

Although not much code is being generated by the system because it can use many high level building blocks (see the Code panes in the various stages of the design process), having 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 determines their correct order. The system knows what types of icons are available, how an icon is associated with a window, etc..

User interface techniques like prompting and menus make it easy to experiment in the domain of window construction. The system makes sure that errors are "impossible". This does not mean that these techniques make sure that users always *understand* what they are doing. A test person, for example, could not tell the difference between the "`add-title:`" and "`specify-title:`" suggestions before actually trying them. Similarly, the pop up menu of available icon types does not show what the icons look like (it should be replaced by a pictorial type of menu). On the other hand, we claim that it is not appropriate to invest too much to make sure that a satisfactory result is achieved with the first try. Rather, the system is intended to support an experimental style and the "`undo:`" operation should make it easy

to step back and retract a decision and to proceed differently. So far, the UNDO feature is not 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: removing the title bars of a window implies that the buttons have to be removed too.

A previous version of the system required the user to know which button icons and which messages are available for adding to the title bar, and they had to be specified by name. The current version with the "`Select a Message`" menu (Figure 6-11) shows some choices to select from and also allows to type in the name of a message that is not displayed, possibly because it is some "internal" message or because it is a new message that the user is going to define later. This method gives users some guidance but does not prevent them from entering unforeseen values.

Methods like this can quite easily be applied in well structured domains like the present one. There are two problems, however, that need to be addressed. The first one is the understanding problem. Seeing an option in a menu does not imply that its significance is obvious. Why means "`associate-icon:`"? What is the function of a window's icon? Another problem may be the sheer number of options. We did not look into this problem because it does not occur in this relatively small system, but future systems may offer hundreds of choice points. For these design kits, a system of reasonable defaults may provide some help if it is combined with a set of predefined samples that are already rather specific starting points.

With the current implementation it is not easy 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 pieces of the code and obtain an explanation of its function and which user action created it. Many questions are still open: 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 abstraction gap "class - instance" constitutes 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 them. Properties cannot be specified as immediate values but have to be specified as defaults or methods inherited by instances. If the user, experimenting with an instance of the window type, changed 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 really used. 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. Users can see how window types are constructed from predefined components. They learn that defining a new window type means creating a new class that inherits from some existing classes and is augmented with new defaults, slots, and methods. Therefore, the system provides a good starting point for learning about the concepts and structure of this domain.

6.3 TRIKIT

Characterization of the Problem Situation: A very common user interface problem is the display and modification of hierarchical and network structures. Application systems which deal with structures of databases, directory trees, or dependency graphs are examples.

Approach: For this purpose, a design kit for graph display/edit tools has been built. Figure 6-12 illustrates its usage. The application programmer who is an expert for the application system, but not for building user interfaces, 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 to display and edit the underlying data structure.

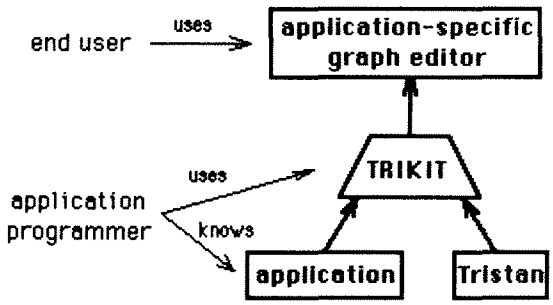

Figure 6-12: Usage of TRIKIT

6.3.1 Application Domain of TRIKIT

Many data structures of computer programs can be viewed as graphs. The nodes are data items which are interconnected with arcs representing a semantic relationship between them. In the following we will use the example of a hierarchical file system which may be displayed as in Figure 6-13. Here the nodes are directories and files, which are the leaf nodes. The arcs represent the membership relation between files and directories (which can themselves be members of other directories). Each one of the nodes is a data structure with properties like name, creation time, owner, protection, size. TRIKIT requires some access functions to these data structures. In this example there are functions for retrieving pieces of the graph (e.g., `list-directory`), and for creating and deleting nodes and arcs.

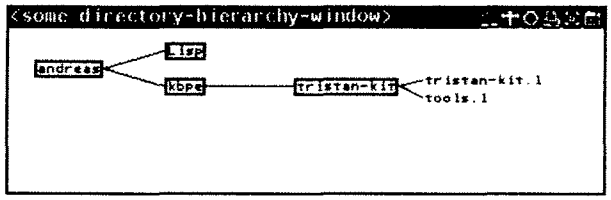

Figure 6-13: A hierarchical file system display

Also, there is a way to refer to particular nodes of the structure by a name relative to some "current directory" or by an absolute path name specifying the way from the/a root of the hierarchy, or by some other description. Conversely, a screen representation must be defined for the nodes. This 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.

6.3.2 The Generic Graph Display/Editor

TRIKIT is based upon Tristan [Nieper 85], a generic display and editing system for graph structures through direct manipulation on the screen. Tristan provides facilities for:

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

Tristan is independent of the particular node representation. It only assumes it to be a subclass of a certain general window class.

6.3.3 Description of TRIKIT

TRIKIT presents itself to the user as an interaction sheet as shown in Figure 6-14 (top window). It is the place where the user specifies the interface to the application, chooses a graphical representation for the nodes, and controls the creation of the user interface.

The following types of fields may be found in the interaction sheet (some of them were also used in WLISPRC, see Section 6.1):

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 (an ObjTalk inheritance hierarchy display system[2]; Figure 6-14: bottom window). This allows the user to familiarize him/herself with it, to modify parameters and to find out about their significance.

Clicking the square representing the `example item` subform produces the form of Figure 6-15. 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 using the example of building a directory editor like the one shown in Figure 6-13. A directory editor is a tool to view a hierarchical file system and to do operations on it such as creating/removing a directory, moving a file into another directory, and renaming files.

---

[2]It was used to create Figure 6-2.

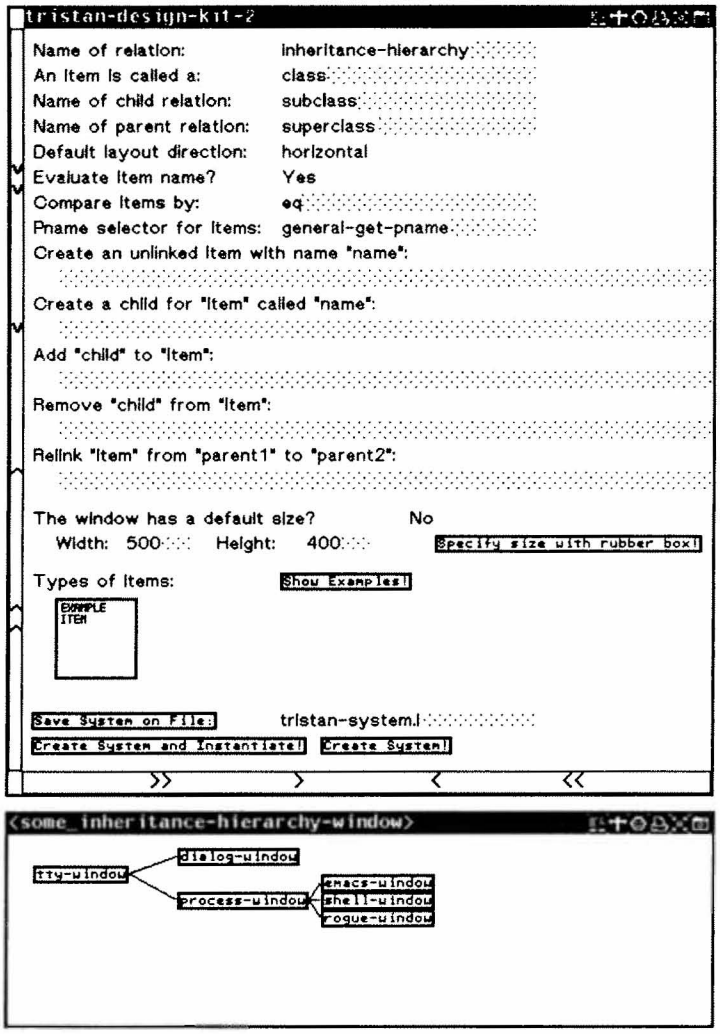

Figure 6-14: Initial state of the main form and an inheritance hierarchy window generated from it

In Figure 6-16 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, of the items that are elements of the relation, and of the links to superordinate and subordinate nodes in the relation. The next field (`Evaluate item name?`) says that a 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 directories. No other changes have been made to this form.

The `example item` form has been renamed (Figure 6-17) and the most important fields have been adapted to the new application:

- the `de:parents` function computes the list of superdirectories;
- the `de:children` function computes the subdirectories, i.e., the contents of the directory;
- the `de:pname` function in the label field computes a "print name", a label, for the items, i.e., it strips off the leading pathname component and leaves the file name which is unique only locally within its directory.

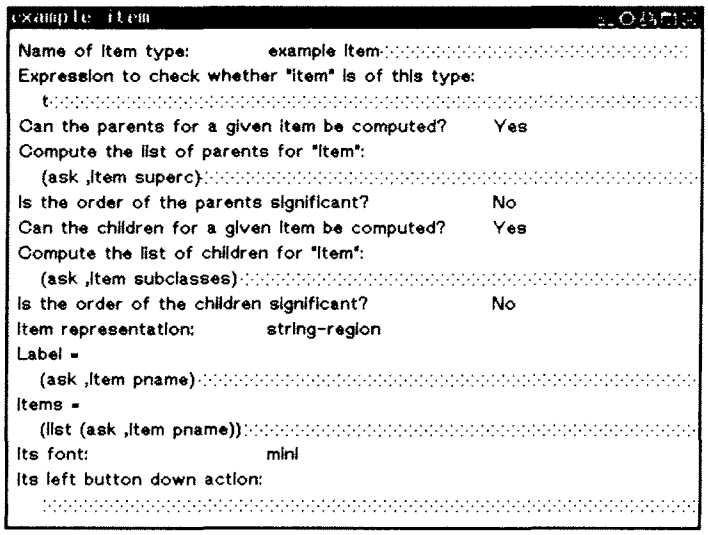

Figure 6-15: Initial state of the node form

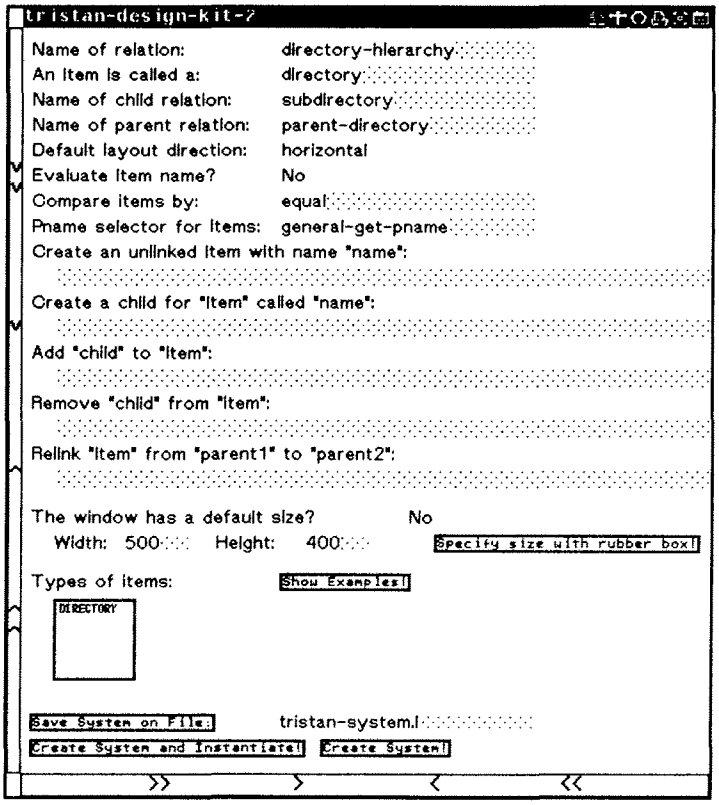

Figure 6-16: Main form, modified to describe a directory editor

Functions with a "de:" prefix (`de:parents`, `de:children`, `de:pname`) belong to the application domain. They are application specific and have to be supplied by the user (the application expert).

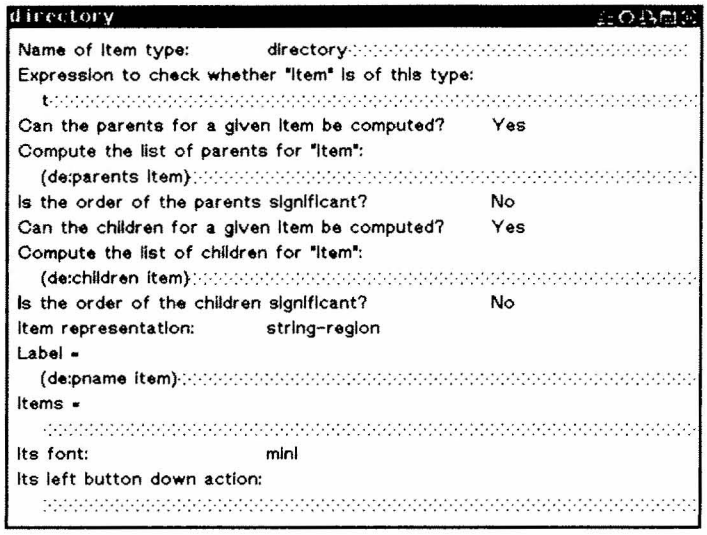

Figure 6-17: Node form, modified to describe a directory node

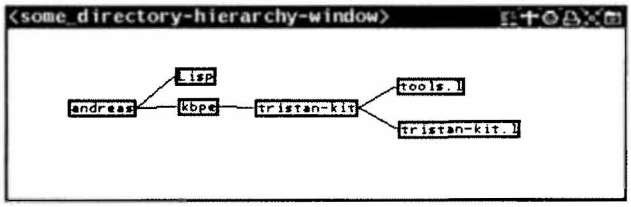

Figure 6-18: An example directory hierarchy window

The mentioned modifications are sufficient to produce a first, 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.

In Figure 6-18 this instance can be seen showing a directory hierarchy. There is 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 it.

Figure 6-19 shows on the left the main menu of directory hierarchy windows. In addition to generic window operations such as "`move`" or "`kill`", three application specific operations have been added automatically:

`replan-layout`
: supplied by the Tristan system: an operation to automatically rearrange the layout of the graph;

`display-subhierarchy...`
: also supplied by the Tristan system: display all the (recursively) subordinated nodes of a given node;

`display-directory...`
: generated by TRIKIT: display a given directory (this is a renamed version of "`display-item...`" of Tristan).

The trailing triple points indicate that a name is going to be prompted for by the system.

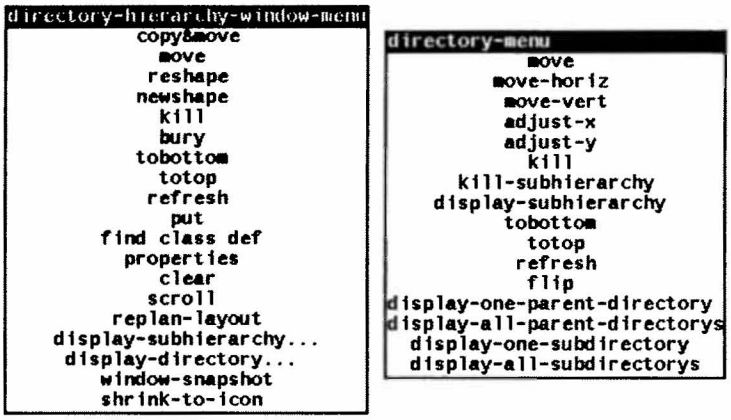

Figure 6-19: The main menus of the directory hierarchy window and the directory node

The menu on the right of Figure 6-19 is associated with directory nodes. Again, some of the operations are general (like `move`), others are supplied by Tristan (`kill-subhierarchy`) and the display operations are generated by TRIKIT.

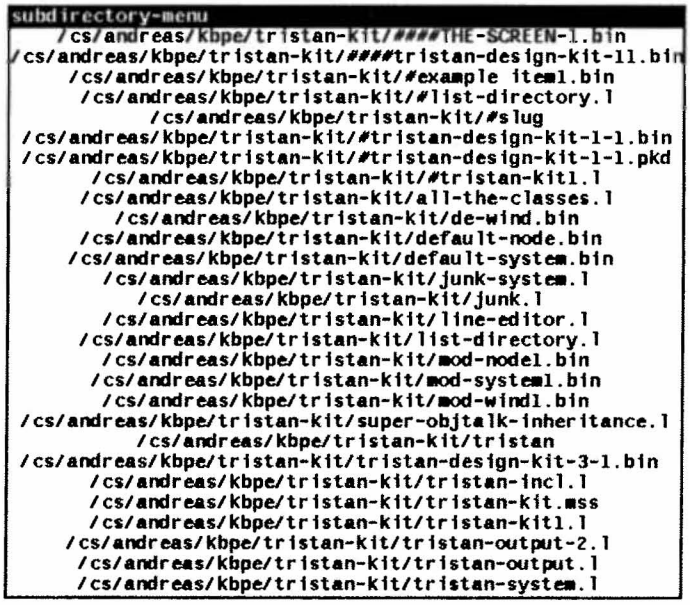

Figure 6-20: The subdirectory menu

If, for example, the directory menu has been popped up on the node labeled `tristan-kit` and `display-one-subdirectory` has been selected, the menu in Figure 6-20 will pop up. Because the system knows how to compute all the subordinate nodes of a given node (Figure 6-17) it is able to determine what should go into this menu.

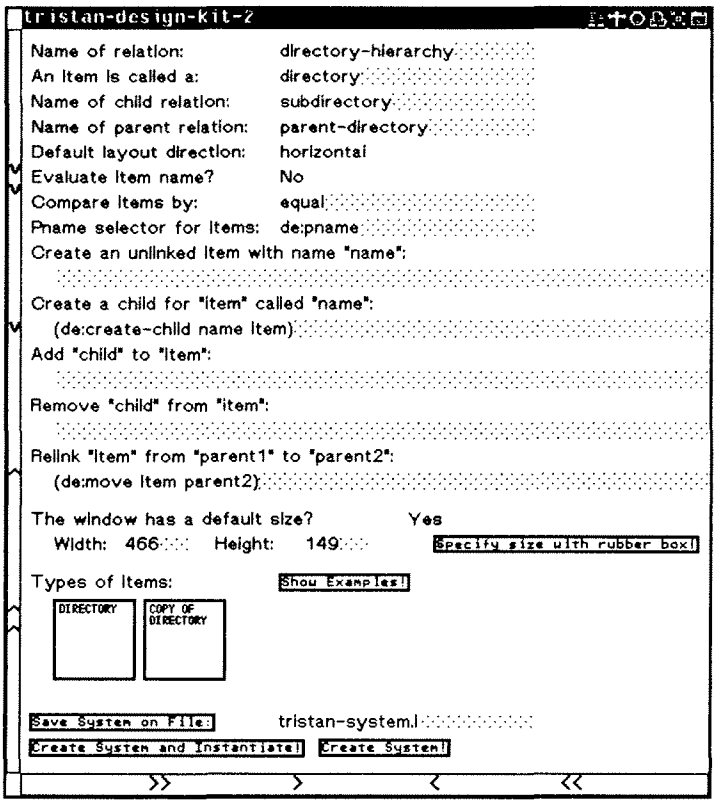

Figure 6-21: Extended description of the directory hierarchy window

The directory hierarchy editor, as it is, has a number of shortcomings. One of them certainly is the large subdirectory menu. It would be nice to display the file names without the leading path names (`/cs/fischer/kbpe/tristan-kit/`). This can be achieved by specifying a special pname (print name) selector for the menus. In Figure 6-21 the pname selector field has been changed to `de:pname`. Also, in order to make the system into a true editor it should be possible to create new nodes and to alter the graph structure. For this purpose, the meaning in terms of the application of creating a child (`de:create`) and of relinking a node from one parent to another (`de:move`) has been specified in the main form.

Also, there are actually two types of nodes in the application: directories and files. Therefore, the user creates a new subform (`Copy of Directory`, Figure 6-21) by cloning the existing one. Now, it is necessary to tell the system how to distinguish between the two node types. Consider Figure 6-22. 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 6-23). Instead an action to load the file into an editor has been specified. Since the nodes are different in their semantics it would be nice to display them differently. The `Item representation` field has been set to `label-region` which has no frame. Figure 6-24 shows a window created according to these modifications.

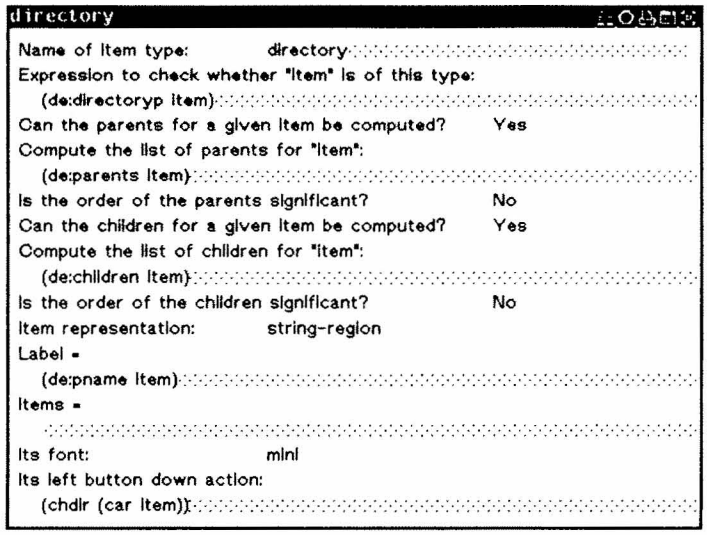

Figure 6-22: The directory node form

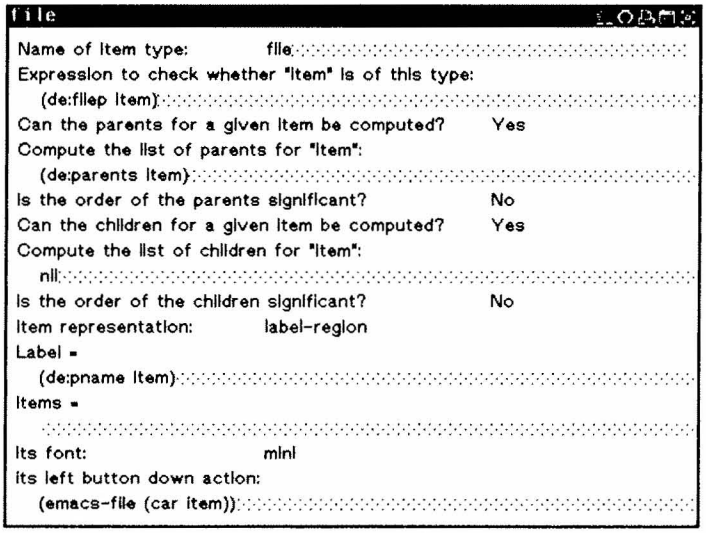

Figure 6-23: The plain file node form

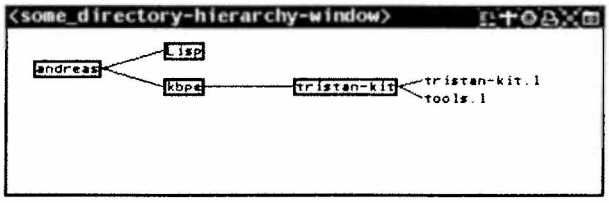

Figure 6-24: A directory hierarchy window according to the modified description

The directory menu of the modified system (Figure 6-25) shows the two new operations

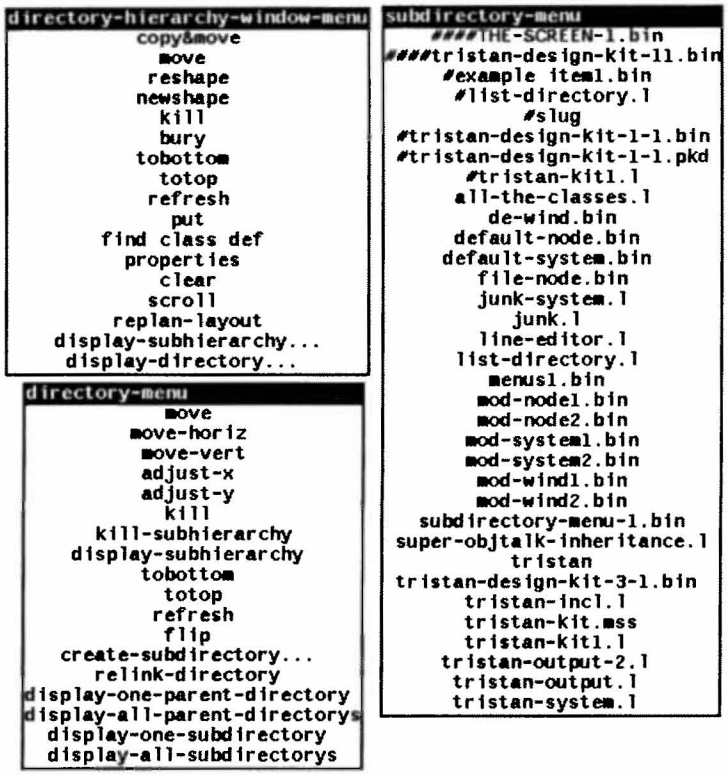

Figure 6-25: The new menus of directory hierarchy window and directory node

(`create-subdirectory...` and `relink-directory`) that have become possible through the modifications in the main form (Figure 6-21). Also, the subdirectory menu shows the file names without their paths as returned by the `de:pname` function.

In case this version of the system does not yet satisfy the requirements of the user, he/she will have to descend one level and work on the system on the implementation level. The `Save System on File` operation in the main form creates a file containing the code of the directory editor, which then may be used as a basis for further extensions (see Appendix I for more details).

6.3.4 A Rule Dependency Display: Experiences with TRIKIT

This section describes the application of TRIKIT to the visualization of the relationship of rules of an expert system. A rule would be displayed as a small rectangle with the rule number in it and, on its left, the premises and on the right the conclusions, both as larger rectangles with several lines of text.

Separate rules can share nodes within this network. For example, the conclusion of one rule could be necessary for a clause of the premise of another rule to be true. This display was very helpful to have an overview of and to debug a set of approximately 100 rules.

When using the system, problems with certain features and concepts became apparent which were not initially understood:

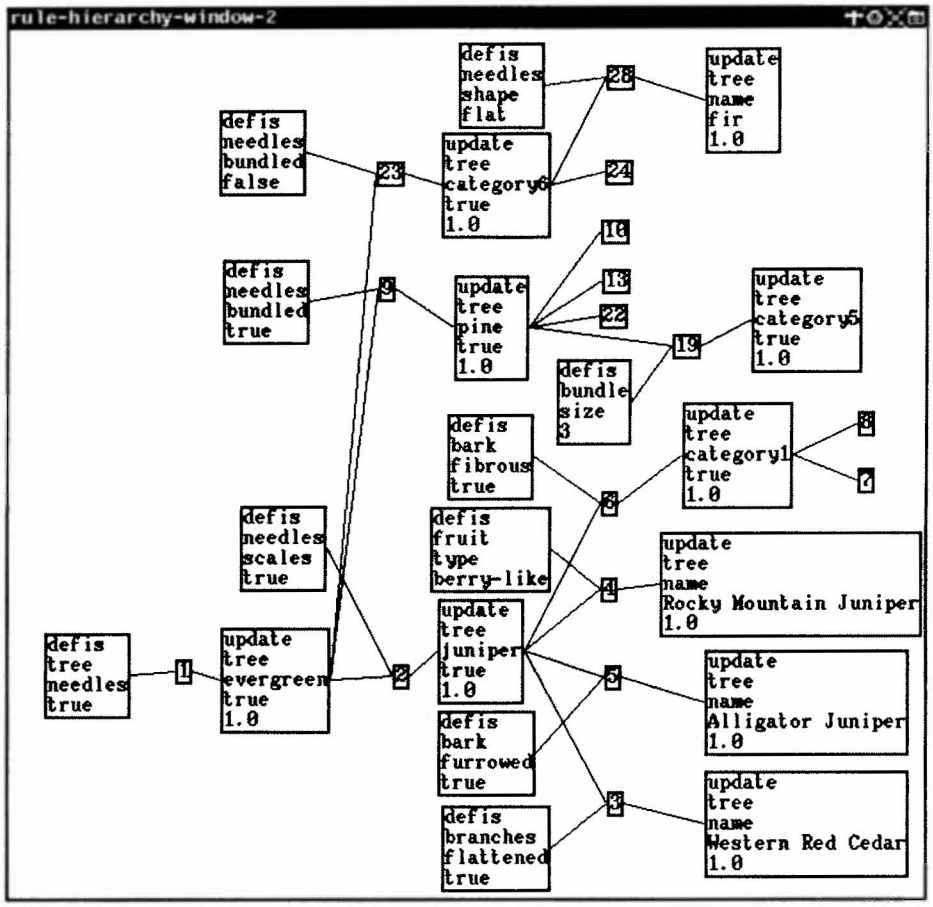

Figure 6-26: A rule dependency display

- The system can be: created, created and instantiated, and saved. The user needs to understand what is being created and what is saved (e.g., a definition of a hierarchy? the forms? or his application? a specific picture?). Making the objects involved visible, can make the system more transparent.
- It's not obvious at first that you need to use two types of "sheets" to define the relation. The first relates to the hierarchy in general (Figure 6-21). The second one (Figure 6-15), which you get by "clicking" example node, defines the properties of different types of nodes in the system. The titles of the sheets should be changed to better reflect their purpose.
- The system also contains default values in the main hierarchy sheet. The user may not realize this.
- In the sheets the two terms *item* and *name* are used. It did not become clear that *item* refers to the elements of the application relation as data structures, whereas *name* referred to their external representation through an identifier.
- Some parts of the system can be understood through experimentation only. 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 (`Pname selector for items`) is not obvious. Also, users may not know whether a system supplied value of a field (`general-get-pname`) works with their application. Adding more knowledge about the applicability of defaults can solve this problem.

The use of the fields is not obvious. Some values of fields have functional purposes, for example, they specify how to compute the subdirectories; others just have label purposes such as to serve as the title of a menu.

6.3.5 Evaluation

TRIKIT may be used to construct useful systems without knowing details of the selected building blocks. Although the design space is limited by the available options in the forms there is still the possibility of using this system to create a prototype which may be refined on a lower level.

The system has been used to build the following applications:

- ObjTalk inheritance hierarchy editor
- UNIX directory editor
- Subwindow hierarchy display
- A project team hierarchy
- Emycin rule dependency display (see Section 6.3.4)

A critical design decision was to not make the user explicitly aware of the fact that for each item in his or her relation a node object will be created that represents the item on the screen. When setting the font of the item, the user refers to the node object. When specifying how to compute the parents of an item, an element of the application relation is referred to. The system tries to conceal this ambiguity and present a simpler model of itself to the user. Initially this distinction was explicit and led to confusion. It is unclear, if it might turn out to be necessary to reintroduce this distinction, when more functionality is added to TRIKIT.

The design space of possible systems must be extended. This may be done, for example, by extending the forms with new fields and by providing new forms for other aspects or views of the system. The tool could be extended to:

- visualize dynamic processes by highlighting currently involved nodes (inferences in the rule display),
- allow to send output to nodes,
- support more than one type of links between nodes,
- be integrated with efforts to build a more general window design kit (see Section 6.2). This could give the application programmer more control over the behavior of the graph window as well as the individual nodes. The generated graph display/editor could be made a part of a larger system.

7. Experiences with Methodologies and Systems to Support Constrained Design Processes

Advantages. We have used the design kits described in the previous section for some time and they have been useful in the following ways:

- They increase the control of the user over systems without making it necessary for the user to learn many details. This became apparent when casual users started using WLISPRC instead of relying on standard initialization files to tailor their systems manually.
- They acquaint users with new and complex system features (e.g. WIDES).
- Systems can be created more quickly, because one can rely on well-developed parts and one can take advantage of stable subassemblies (e.g., exploiting the rich inheritance network in the WLISP system). Simon [Simon 81] demonstrates that this is a crucial aspect in the development of complex systems.

Who uses the design kits? Our experiences have 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 (see the use of TRIKIT to create the rule dependency display in the previous section).

Human-Computer Interaction. It is misleading to assume that knowing how to use a design kit would come for free and would not require a learning process at all. Learning processes are required at different levels in using design kits: users have to operate on different descriptive levels, they need to understand the domain concepts used in the kit 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, at the same time, retaining or even improving the simplicity of use. WIDES is currently easy to use -- but will we be able to retain this when the system will cover more kinds of objects (e.g., menus, icons, gauges) and not only windows?

A difficult question has been to establish a shared vocabulary between the designer and the user to enable the user to understand the descriptions for the required inputs. The label `Pname selector for items` (Figure 6-21) is not obvious to someone who does not know this technical term of the menu system. To remedy some of the shortcomings, we have to solve the following problems:

- try to find a better conceptualization of the design task and use it to restructure the forms to make them easier to understand;
- use more carefully selected examples to convey a better feeling of what needs to be done (by taking task structures and the user's knowledge into account);
- provide optional descriptions 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: the information about creating a link between two nodes should be asked for only at the first time this action is being executed.

Currently the interaction with the design kits is mostly a *monologue* by the user. The system does not actively act on user inputs. Our goal is to make *dialogues* possible in which the user and the system take turns with their actions, 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 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 it assumes that it is going to be more like that one than like the original defaults.

Consistency. Once a system has been modified at the code level, it is no longer possible to use the design kits to make further modifications. A program analysis component could make it possible for the high level (form) description and the program code to coexist. In this way the user could use both languages alternately.

A set of issues needs to be further explored in connection with the conceptual distance between a description and what it describes. What happens to an existing object whose description has been changed? Should it be updated? This is not always desired or possible, because

- immediate updates may be computationally too expensive,
- the screen display may change to a degree that the user loses track of where things are,
- changes of descriptions of actions executed at the creation time of objects have no effect on already existing objects (e.g., initial size of a window),
- in cases where properties are not visible, they can not be changed in a direct manipulation style.

Methodologies and Tools for the Designer of Design Kits. Regarding design kits as desirable components of computer systems, the question arises: what do we have to do to develop *kit-kits*, i.e., design kits for designers of design kits. What challenges and problems do designers face in developing design kits? Their design requires a qualitatively different description level (using many more abstractions) and they have to defer commitment so users are left with their share and influence on the design.

8. Comparison with other Systems

The problems adressed by our research are widespread and there exist a large number of research efforts to address them. We briefly describe some other work which is most closely related to our approach.

KBEmacs [Waters 85; Waters 86] is an effort to produce a prototype of the Programmer's Assistant [Rich, Shrobe 78]. The system's goal is to make expert programmers superproductive by having the assistant carry out the mundane and repetitive parts of the programming task. KBEmacs contains a small library of algorithmic cliches (serving the same role as the classes in WIDES), which provide a vocabulary of relevant intermediate and high-level concepts and the system is able to deal with two representations of a program (a textual representation and a plan representation). Based on the currently small number of cliches, problems of their organization and of how a programmer will find the right kind of cliche have not been relevant for this research effort.

Trillium [Henderson 86] is a computer-based design environment to support rapid prototyping of user interfaces to simple machines like copiers and printers. It provides a construction kit of elements to build interfaces quickly, so the designer can shift quickly between designing the interface and trying it out to evaluate the effects achieved. By restricting the problem domain, Trillium provides a powerful tool which is in wide-spread use. It remains to be seen, how some of the ideas explored in this work (e.g., support for constrained design processes like composition and specialization) will carry over to other design environments.

User Interface Management Systems [Olsen et al. 84] (e.g., the COUSIN system [Hayes, Szekely, Lerner 85]) are based on the idea that the user interface portion of a program can be separated from the portion implementing its functionality. Limiting the information exchange between user interface and application system is a reasonable approach for some problems. Based on our architecture and on the kinds of problems we try to solve (building intelligent support systems such as help, documentation and explanation systems), we claim that a strong separation between interface and application is impossible because the user interface has to have extensive access to the state and actions of the application system. We regard and model a computational system as a collection of communicating objects or agents, each of them having an internal state and an external view, and we provide mechanisms, such as ObjTalk's constraints, to maintain their consistency.

All these systems have *in common* that they attempt to overcome the tediousness of programming, to take advantage of existing components, to speed up the design cycle and to provide an escape mechanism to lower levels if the design abstractions are not contained in the construction kit. To provide this kind of assistance, all of these system are knowledge-based. The **uniqueness of our approach** is the development of a general framework for constrained design processes and the augmentation of construction kits with design kits. Design kits guide the designers when they create a new system; for construction kits with hundreds and thousands of elements, the existence of a design kit is not a luxury but a necessity, especially for designers which are not totally familiar with the construction kit. To make this approach viable, the system must have an understanding about the design space; it must indicate which opportunities and restrictions exist at a certain point in the design process. This is captured to some extent in the suggestion list of WIDES or in the goal browser of the PRIDE system [Bobrow, Mittal, Stefik 86]. Another specific feature of our approach is the effort to integrate design kits into the architec-

ture of intelligent support systems (Figure 3-2). This architecture is a first step towards our long ranging goal to regard programming not as writing code, but as representing knowledge about specific domains. Different system components (e.g., help, documentation, critics, design and visualization support) are then generated in a coherent way from the underlying knowledge base.

9. Conclusions

Give humans some fish and they can eat for a day -- teach them to fish and they have something to eat for their whole life!
Asian Proverb

Our research is concerned with how computer systems can be made more convivial. We would like to identify methods to give users the amount of control over systems that they desire. This is especially important because the number of those systems increases where the *vision* of the designer and the *real needs of the users* are dramatically different. Some of these discrepancies can be reduced by having a heavy user involvement and participation in all phases of the development process, but they will not be eliminated: modifiability by users will be a necessity to cope with new and unforeseen situations. Our research so far has raised the interesting issue that systems which give control to the user might be *less complicated*, because they do not have to anticipate all possible futures (laws are complicated because they cannot be adapted dynamically). We have tried to show some approaches that find new middle roads between the full generality of programming languages and the limitations of turn-key systems. Intelligent support systems that allow the user to carry out constrained design processes are a promising step towards the goal of making computer systems more convivial.

Acknowledgements. The research described here was supported by the University of Colorado, the Office of Naval Research (contract number: N00014-85-K-0842), Triumph Adler (Nuernberg, Germany), and the German Ministry for Research and Technology.

Appendix I. Code Generated by TRIKIT

Compared to the other design kits, a considerably larger amount of code is being generated by TRIKIT. The following listing of the code generated for the directory editor application illustrates the way in which this is done:

1. Code templates are instantiated and filled in from the forms, and
2. specific code is generated for distinguishing between the different node types.

Most of the code is written in the object-oriented language ObjTalk and FranzLisp. It makes use of predefined objects and libraries of the WLisp system.

The comments in the following program code are put in manually and are not generated by TRIKIT. It would be straightforward, however, to generate the comments automatically since they don't use special domain knowledge. Underlined pieces of code are taken directly from the forms (Figures 6-21, 6-22 and 6-23).

```
;;; Load the Tristan library
(include-file 'tristan-incl '/cs/fischer/lib/system85/trikit/)

;;; The directory editor window type
(ask class renew: |file system-window|
  (methods
    (get-item: ,?name => (progn (de:create-item name)))

    ;; find out type of item and create the right type of node
    (create-node-for: ,?item =>
      (cond ((de:directoryp item)
             (ask directory instantiate:
               (item = ,item) (label = ,(de:pname item))))
            ((de:filep item)
             (ask file instantiate:
               (item = ,item) (label = ,(de:pname item))))
            (t (error '|file system-window:|
                 "Don't know how to create a node for:" item))))

    ;; renamed method: items are files in this application
    (display-file...: => (ask self display-item...:)))

  (descr (top-menu (default |file system-window-menu|)) ;defined below
         (item-name (default 'file))
         (background (default nil))
         (layout-direction (default 'horizontal))
         (equality-operator (default 'equal)))
  (superc window-repr-mixin
          window-util-mixin
          hierarchy-mixin
          scroll-hand-mixin
          super-scroll-window))

;;; Toplevel menu of the directory editor window
(ask pop-up-menu remake: |file system-window-menu| with:
  (pname-selector = lmenu:get-pname)
  (items = (;; general window operations
```

```
            copy&move move move-out reshape newshape kill bury
            tobottom totop
            refresh bgupload |find class def| shrink-to-icon
            window-snapshot
            clear scroll | |
            ;; operations provided by Tristan
            replan-layout display-subhierarchy...
            ;; operations generated by TRIKIT
            display-file...)))

;;; The node type for representing files
(ask class renew: file
  (methods
    (left-button-down: ,?rep =>=>
      (let ((item ,!item)) (emacs-file (car item))))
    (get-parents-items: =>
      (let* ((item ,!item)
             (parents (de:parents item)))
        (or (null parents) (dtpr parents)
            (error "The list of parents is required. "
                   "Check the \"Compute the parents\" field!"))
        parents))

    ;; renamed methods
    (display-one-parent-directory: => (ask self display-parent...:))
    (display-all-parent-directorys: => (ask self display-parents:))

    (get-children-items: =>
      (let* ((item ,!item)
             (children nil))
        (or (null children) (dtpr children)
            (error "The list of children is required. "
                   "Check the \"Compute the children\" field!"))
        children))

    ;; renamed methods
    (display-one-member: => (ask self display-child...:))
    (display-all-members: => (ask self display-children:)))

  (descr (top-menu (default file-menu)) ;defined below
         (parents-menu (init parent-directory-menu))
         (children-menu (init member-menu))
         (font (default 'mini)))
  (superc node-repr-mixin
          node-util-mixin
          node-mixin
          adaptive-text-region))

;;; Operations on file nodes
(ask pop-up-menu remake: file-menu with:
  (pname-selector = lmenu:get-pname)
  (items = (;; general window operations and operations
            ;; provided by Tristan
            move move-horiz move-vert adjust-x adjust-y kill
             kill-subhierarchy
            display-subhierarchy tobottom totop refresh
             insert-link-to-child
```

```
            remove-link-to-child flip | |
            ;; operations generated by TRIKIT
            create-member... relink-file
            display-one-parent-directory
            display-all-parent-directorys
            display-one-member display-all-members)))

;;; The node type for representing directories
(ask class renew: directory
  (methods
    (left-button-down: ,?rep =>=>
      (let ((item ,!item)) (de:make-current item)))

    (get-parents-items: =>
      (let* ((item ,!item)
             (parents (de:parents item)))
        (or (null parents) (dtpr parents)
            (error "The list of parents is required. "
                   "Check the \"Compute the parents\" field!"))
        parents))

    (display-one-parent-directory: => (ask self display-parent...:))
    (display-all-parent-directorys: => (ask self display-parents:))

    (get-children-items: =>
      (let* ((item ,!item)
             (children (de:children item)))
        (or (null children) (dtpr children)
            (error "The list of children is required. "
                   "Check the \"Compute the children\" field!"))
        children))

    (display-one-member: => (ask self display-child...:))
    (display-all-members: => (ask self display-children:)))

  (descr (top-menu (default directory-menu)) ;defined below
         (parents-menu (init parent-directory-menu))
         (children-menu (init member-menu))
         (font (default 'mini)))
  (superc node-repr-mixin
          node-util-mixin
          node-mixin
          adaptive-text-region-with-border))

;;; Operations on directory nodes
(ask pop-up-menu remake: directory-menu with:
  (pname-selector = lmenu:get-pname)
  (items = (;; general window operations and operations
            ;; provided by Tristan
            move move-horiz move-vert adjust-x adjust-y kill
            kill-subhierarchy
            display-subhierarchy tobottom totop refresh
            insert-link-to-child
            remove-link-to-child flip | |
            ;; operations generated by TRIKIT
            create-member... relink-file
            display-one-parent-directory
```

```
            display-all-parent-directorys
            display-one-member display-all-members)))

;;; Menu used to select a parent directory
(ask pop-up-menu remake: parent-directory-menu with:
  (pname-selector = de:pname)
  (items = nil))

;;; Menu used to select a directory member
(ask pop-up-menu remake: member-menu with:
  (pname-selector = de:pname)
  (items = nil))

;;; Make the directory editor known to the system
(ask window-types add: |file system-window|)
```

References

[Bobrow, Mittal, Stefik 86]
D.G. Bobrow, S. Mittal, M.J. Stefik, *Expert Systems: Perils and Promise*, Communications of the ACM, Vol. 29, No. 9, September 1986, pp. 880-894.

[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, 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.

[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.

[Fabian, Lemke 85]
F. Fabian Jr., A.C. Lemke, *WLisp Manual*, Technical Report CU-CS-302A-85, Department of Computer Science, University of Colorado, Boulder, CO, February 1985.

[Fischer 81]
G. Fischer, *Computer als konviviale Werkzeuge*, Proceedings der Jahrestagung der Gesellschaft fuer Informatik (Muenchen), Springer-Verlag, Gesellschaft fuer Informatik, Berlin - Heidelberg - New York, 1981, pp. 407-417.

[Fischer 86]
G. Fischer, *From Interactive to Intelligent Systems*, in J.K. Skwirzynski (ed.), *The Challenge of Advanced Computing Technology to System Design Methods, Proceedings of a NATO Advanced Study Institute (University of Durham)*, Springer-Verlag, Berlin - Heidelberg - New York, 1986, pp. 185-212.

[Fischer, Boecker 83]
G. Fischer, H.-D. Boecker, *The Nature of Design Processes and how Computer Systems can Support them*, Integrated Interactive Computing Systems, Proceedings of the European Conference on Integrated Interactive Computer Systems (ECICS 82), P. Degano, E. Sandewall (eds.), North-Holland, 1983, pp. 73-88.

[Fischer, 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, CO, 1986.

[Fischer, Lemke, Rathke 87]
G. Fischer, A.C. Lemke, C. Rathke, *From Design to Redesign*, Proceedings of the 9th International Conference on Software Engineering (Monterey, CA), IEEE Computer Society, Washington, D.C., March 1987, pp. 369-376.

[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, FL), IEEE Computer Society, Los Angeles, CA, March 1984, pp. 358-368.

[Furuta, Scofield, Shaw 82]
R. Furuta, J. Scofield, A. Shaw, *Document Formatting Systems: Survey, Concepts and Issues*, Communications of the ACM, Vol. 14, No. 3, September 1982, pp. 417-472.

[Goldberg 81]
A. Goldberg, *Smalltalk*, BYTE, Special Issue, Vol. 6, No. 8, 1981.

[Gosling 82]
J. Gosling, Carnegie-Mellon University, *Unix Emacs*, Pittsburgh, 1982.

[Hayes, Szekely, Lerner 85]
P.J. Hayes, P.A. Szekely, R.A. Lerner, *Design Alternatives for User Interface Management Systems Based on Experience with COUSIN*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), ACM, New York, April 1985, pp. 169-175.

[Henderson 86]
D.A. Henderson, *The Trillium User Interface Design Environment*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 221-227.

[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, Hillsdale, NJ, 1986, pp. 87-124, Ch. 5.

[Illich 73]
I. Illich, *Tools for Conviviality*, Harper and Row, New York, 1973.

[Kay 84] A.C. Kay, *Computer Software*, Scientific American, Vol. 251, No. 3, September 1984, pp. 52-59.

[Lemke 85]
A.C. Lemke, *ObjTalk84 Reference Manual*, Technical Report CU-CS-291-85, Department of Computer Science, University of Colorado, Boulder, CO, 1985.

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

[Norman 86]
D.A. Norman, *Cognitive Engineering*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 31-62, Ch. 3.

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

[Rich, Shrobe 78]
C. Rich, H.E. Shrobe, *Initial Report on a Lisp Programmer's Apprentice*, IEEE Transactions on Software Engineering, Vol. SE-4, No. 6, 1978, pp. 456-467.

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

[Stallman 81]
R.M. Stallman, *EMACS, the Extensible, Customizable, Self-Documenting Display Editor*, ACM SIGOA Newsletter, Vol. 2, No. 1/2, 1981, pp. 147-156.

[Waters 85]
R.C. Waters, *The Programmer's Apprentice: A Session with KBEmacs*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1296-1320.

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