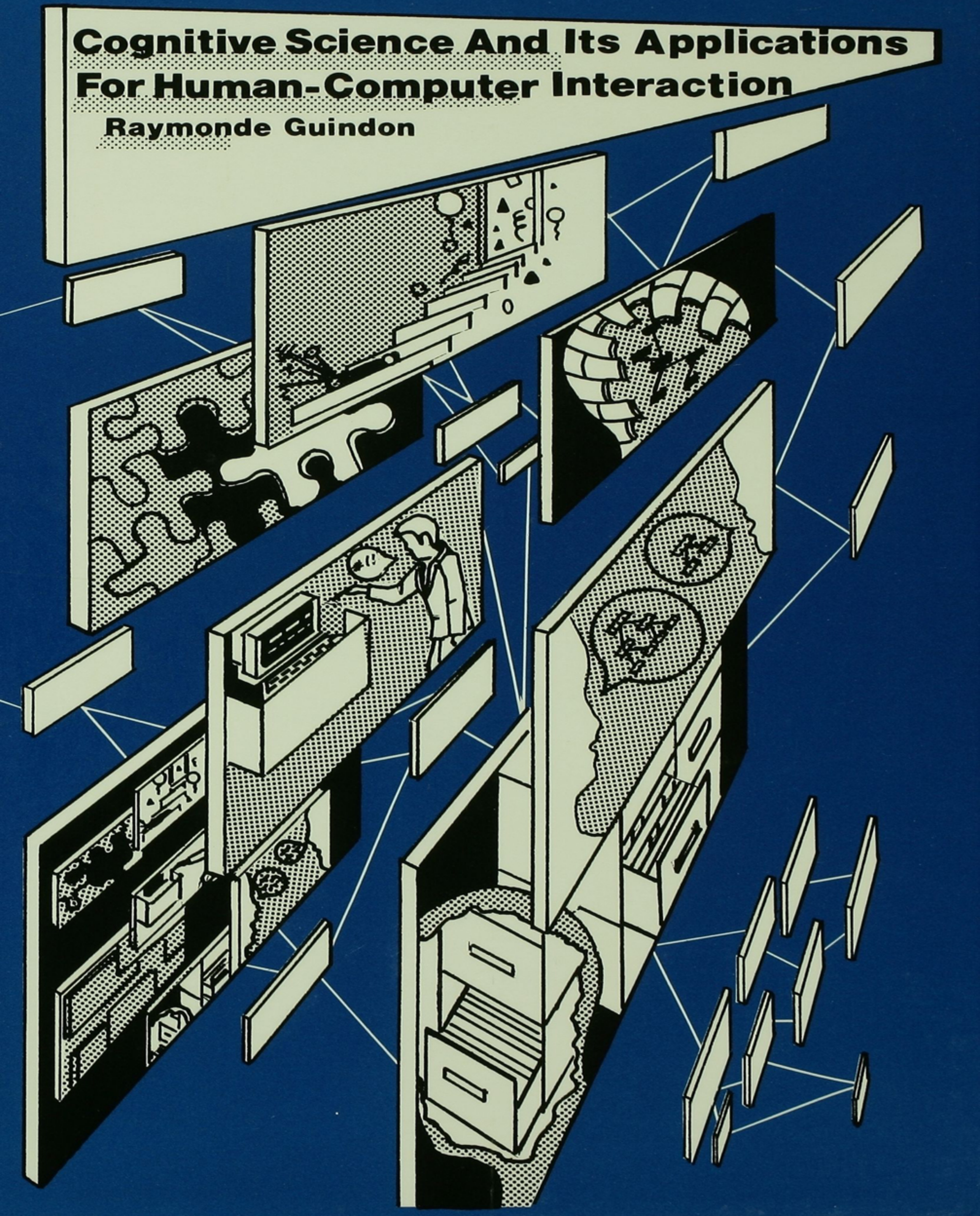
Cognitive Science And Its Applications For Human-Computer Interaction

Raymonde Guindon



COGNITIVE SCIENCE and its APPLICATIONS for HUMAN-COMPUTER INTERACTION

COGNITIVE SCIENCE and its APPLICATIONS for HUMAN–COMPUTER INTERACTION

Edited by

Raymonde Guindon

Microelectronics and
Computer Technology Corporation

LAWRENCE ERLBAUM ASSOCIATES, PUBLISHERS
1988 Hillsdale, New Jersey Hove and London

Lawrence Erlbaum Associates, Inc., Publishers
365 Broadway
Hillsdale, New Jersey 07642

Library of Congress Cataloging-in-Publication Data

Cognitive science and its applications for human-computer interaction.

Includes indexes.
1. Interactive computer systems. 2. System design.
3. Cognition. I. Guindon, Raymonde.
QA76.9.I58C63 1988 004'.33 87-36523
ISBN 0-89859-884-2

Printed in the United States of America
10 9 8 7 6 5 4 3 2

Contents

Contributors

Kim M. Fairchild
Microelectronics and Computer
Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

Gerhard Fischer
Computer Science Department
University of Colorado
Boulder, Colorado 80309

Barbara Fox
Linguistics Department
University of Colorado
Boulder, Colorado 80309

George W. Furnas
Bell Communications Research
435 South Street, Room 2M 397
Morristown, New Jersey 07960

Raymonde Guindon
Microelectronics and Computer
Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

William P. Jones
Microelectronics and Computer
Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

Roger King
Computer Science Department
University of Colorado
Boulder, Colorado 80309

Andreas C. Lemke
Computer Science Department
University of Colorado
Boulder, Colorado 80309

Clayton Lewis
Computer Science Department
University of Colorado
Boulder, Colorado 80309

James E. McDonald
Computing Research Laboratory
New Mexico State University
Las Cruces, New Mexico 88003

Peter G. Polson
Department of Psychology
University of Colorado
Boulder, Colorado 80309

Steven E. Poltrock
Microelectronics and Computer
Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

Roger W. Schvaneveldt
Computing Research Laboratory
New Mexico State University
Las Cruces, New Mexico 88003

Paul Smolensky
Computer Science Department
University of Colorado
Boulder, Colorado 80309

Preface

Breaking the communication barriers between experts in different disciplines requires overcoming differences in jargon, and more importantly, profound differences in paradigms. Being an expert in more than one technical area is a rare achievement. The field of human-computer interaction is striving to provide the conceptual foundations for designing computer tools and the environment needed to perform increasingly more complex and specialized tasks. To achieve this goal, human-computer interaction must rely on the meeting of specialized, expert minds. Each of the research projects presented in this book investigate some critical question on the path of progress in human-computer interaction. These projects would not have been feasible without the multidisciplinarity of the research team or of the researchers themselves.

This book is composed of chapters organized around the theme of multidisciplinary research and the contribution of cognitive science to the research projects. Interestingly, we find instances of research projects overlapping in goals, but using widely diverse methodologies. We also find research projects using the same or similar methodologies to answer quite different questions. These methodologies and techniques come from such diverse fields as scaling and measurement, computer science, experimental psychology, and linguistics. The applications of these varied methodologies and techniques act in synergy to solve the problems posed by human-computer interaction.

Why say in many words what an annotated diagram can explain more concisely and directly? Some of the interconnections between the research projects presented in this book are depicted in the figure on the next page. The dotted links point to the concepts, techniques, and models underlying the research projects, while the solid links point to the goals of the projects.

The goal of Fischer and Lemke, in the first chapter, is to provide maximal access to the rich functionality available in current computer systems to

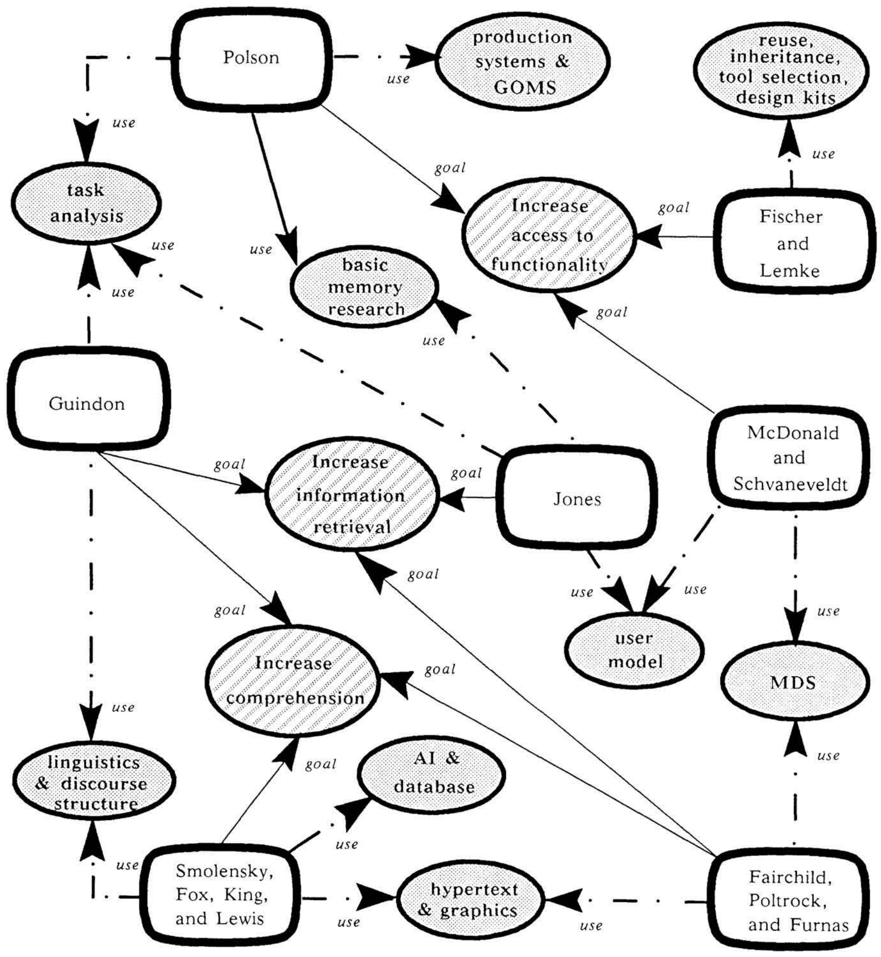

Polson
production systems & GOMS
reuse, inheritance, tool selection, design kits
task analysis
Increase access to functionality
Fischer and Lemke
basic memory research
Guindon
McDonald and Schvaneveldt
Increase information retrieval
Jones
user model
MDS
Increase comprehension
linguistics & discourse structure
AI & database
Smolensky, Fox, King, and Lewis
hypertext & graphics
Fairchild, Poltrock, and Furnas
use
goal

Figure 1: Overlap in goals and techniques between chapters

casual and intermediate users. They present an analysis of these users' needs based on previous research and their own observations. They found that casual and intermediate users are rarely willing or capable of spending the time necessary to acquire the detailed knowledge needed to make adequate use of the available functionality. They have developed a computer environment supporting constrained design processes as a step towards convivial computer systems. These processes allow the novice and intermediate users to access rich functionality without extensive initial learning, while allowing expert users to reuse already developed and tested components. The described constrained design processes are achieved through selection, combination, and instantiation of general tools, and design kits.

In the second chapter, Polson also concerns himself with maximizing the use of available functionality. His strategy is to empirically determine the conditions favoring skills transfer between interfaces. He describes in detail the notion of consistent interfaces in terms of the GOMS model. He builds and validates a model of the transfer process between consistent interfaces based on the cognitive complexity theory. He shows that basic findings from early research in psychology on human memory is relevant to understanding the transfer of skills between computer systems and the phenomenon of interference between interfaces.

Smolensky, Fox, King, and Lewis describe an environment to support reasoning and decision-making. This environment provides for the representation of complex arguments with semi-structured forms. The argumentation language, ARL, captures the formal aspects of the argument, while the user provides the informal components in natural language. The environment, EUCLID, provides some processability of the semi-structured arguments by testing, for example, their well-formedness. It also provides already built-in schemas or templates for many types of arguments. Moreover, users can examine the content and structure of an argument with graphical or tabular displays. EUCLID is expected to increase the logical reasoning skills of users, both in the generation and in the comprehension of arguments and proofs. This research project is based on linguistics research in argumentative discourse structure, artificial intelligence, database technology, and cognitive psychology.

Turning to my chapter, I compare the structure of user-advisor dialogues in their most frequent form, spoken face-to-face between two humans, to the structure of typed dialogues between a user and a computerized advisory system. The purpose of the study is to gather data about users' dialogues to guide the design of natural language front-ends to advisory systems. It is hoped that determining the structure of users' dialogues will help natural language interfaces perform anaphora resolution. The study uses findings and methods from linguistic discourse analysis and experimental psychology to answer a question in human-computer interaction. In the study, a task analysis based on the GOMS model is completed and a corresponding task structure is derived. The task structure seems to have more influence on the structure of spoken face-to-face dialogues than on the structure of typed dialogues. Typed user-advisor dialogues resemble independent queries more than cohesive discourse. Implications for the design of natural languages interfaces are generated.

Contending with information overload, SemNet is a 3-D graphical interface for large knowledge bases. Designed by Fairchild, Poltrock, and Furnas, SemNet allows easy retrieval of information from and exploration of large knowledge bases. The user "travels through" the knowledge base with a choice of many navigation techniques. The knowledge base information is also selectively displayed through FishEye views specified by the user. The positioning and the selection of elements to be displayed is based on techniques such as multidimensional scaling and the centroid heuristic. SemNet has been used and empirically evaluated as an interface to a large knowledge base of Prolog rules to perform morphological analysis.

Also struggling with information overload, Jones' Memory Extender system provides for context based retrieval of files from large filing systems. In the ME system, files and context are represented in an associative network of weighted term links. Three processes underly the adaptive and contextually sensitive retrieval: an exchange of representational information, a decay mechanism, and a spreading activation matching algorithm. The ME system is founded on a task analysis of information retrieval and, as in Polson's model, on basic research on the properties and mechanisms of human memory. By designing a system with good fit to both the task and the user's capabilities, ME provides a computational extension to the user's memory. Jones' research reveals a mutually beneficial interplay between basic research in human information processing and applied efforts to build more usable computer systems.

McDonald and Schvaneveldt share the goal of Fischer and Lemke and of Polson: maximal access to the functionality of computer systems to casual and intermediate users. They propose an interface design methodology which they are testing. The methodology involves uncovering the conceptual models of a computer system from experienced users. These models are uncovered through the use of hierarchical cluster analysis, multidimensional scaling, and the Pathfinder algorithm. They have applied the methodology to the design of an interactive Unix documentation system, Superman II. They present a review of several applications that illustrate key aspects of their methodology.

This book grew from presentations at a regional meeting of the American Association for the Advancement of Sciences, held at the University of Colorado, Boulder. I was asked to help organize a session at the Psychological Sciences section of the conference by Dr. Jesse Purdy. Strong interest was expressed by the conference organizers in the topics of cognitive science and of human-computer interaction. I felt that these topics were quite close to each other in the sense that the development of cognitive science had provided many of the empirical methodologies and conceptual models used in the field of human-computer interaction. So I selected as the theme of the session the contribution of the different disciplines composing cognitive science to the study of human-computer interaction. I also emphasized the multidisciplinarity required of the researchers or teams of researchers working in the area of human-computer interaction. The University of Colorado was an especially appropriate site for the conference because of its excellent Institute for Cognitive Science and multidisciplinary work in human-computer interaction. Moreover, this conference gave me an opportunity to describe some the work in progress in human-computer interaction at the Microelectronics and Computer Technology Corporation.

I wish to thank the many colleagues who reviewed one or more chapters of this book and who significantly helped increase their quality: Ernest Chang, Jeff Conklin, Joyce Conner, Nancy Cooke, Jonathan Grudin, Will Hill, Patrick Lincoln, Gale Martin, Jim Miller, Don Norman, Ken Paap, Nancy Pennington, and Elaine Rich. And thanks to all the contributors who also reviewed each other's chapters and thus, helped produce a more integrated book. Also, this book would have never appeared

without Bill Curtis' encouragement and support throughout. Finally, Joyce Conner performed excellent editorial supervision, accomplished the feat of producing the camera-ready version of this book, keeping track of chapters in various stages of completion, designing the layout of the chapters, ensuring uniformity of style throughout the book, improving the writing style, and much more.

Raymonde Guindon

1
Constrained Design Processes: Steps Towards Convivial Computing

GERHARD FISCHER
ANDREAS C. LEMKE

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 chapter 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, 1986) and to enhance incremental learning processes with knowledge-based systems (Fischer, 1987). In

Section 2 we will briefly describe what we mean 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, 1985; Boecker, Fabian, Lemke, 1985; Fabian, 1986)).
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, 1985) is a generic tool for generating graphical 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 (1973) 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, 1981) 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.

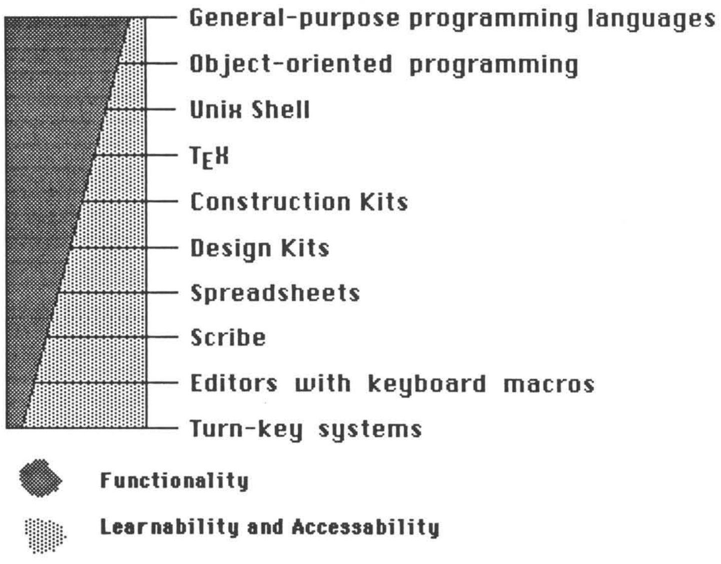

General-purpose programming languages
Object-oriented programming
Unix Shell
TEX
Construction Kits
Design Kits
Spreadsheets
Scribe
Editors with keyboard macros
Turn-key systems
Functionality
Learnability and Accessability

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 (1986)):

   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, 1981) or ObjTalk (Rathke, 1986)), 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 ("keyboard 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.

The user should have control over a tool on multiple levels. Figure 2-3 shows the levels of control for the EMACS editor (Stallman, 1981; Gosling, 1982). The keystroke level together with its special purpose extensions (lisp mode, etc.) is most frequently and most easily used. The lower

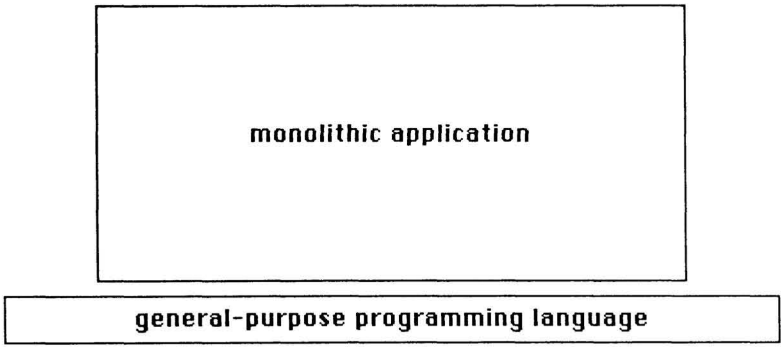


Figure 2-2: Turn-key systems

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.

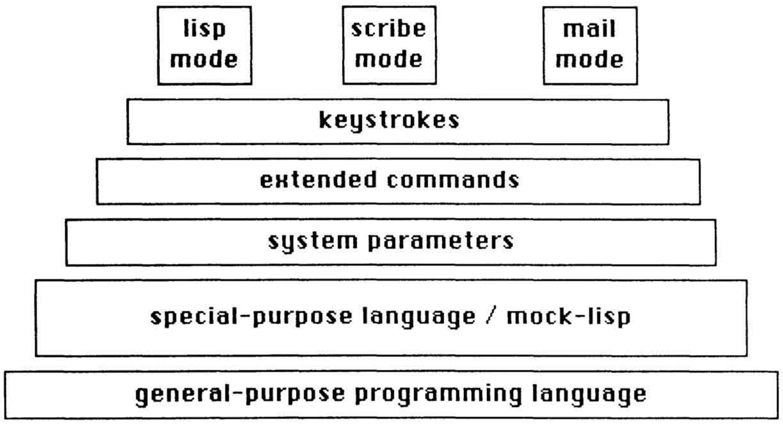


Figure 2-3: Levels of control over the EMACS editor

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. Truly 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 $\TeX$ and Scribe (Furuta, Scofield, & Shaw, 1982), show the following differences (see also Figure 2-1):

1. The $\TeX$ user is viewed as being an author who wants to position objects exactly on the 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 $\TeX$. 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, 1985)) or should systems *be adaptable by the user*? Should systems be composed of simple or intelligent tools (Norman, 1986)? *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, 1983). *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, 1985) 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 be an observer, but should be able to understand what is going on).

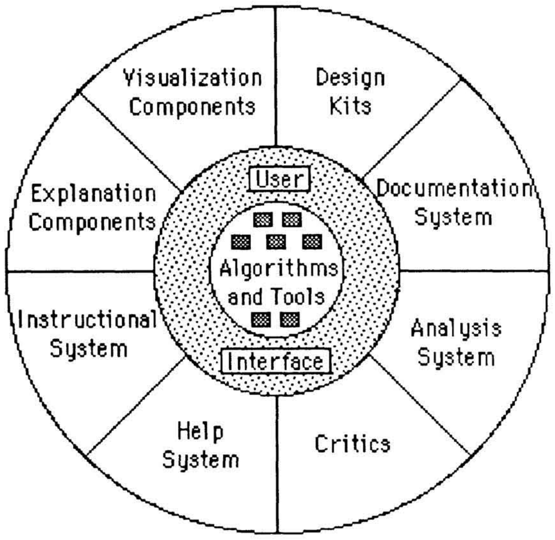


Figure 3-2: The architecture of intelligent support systems

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, 1984),
- help systems in (Fischer, Lemke, & Schwab, 1985),
- critics in (Fischer, 1987),
- visualization tools in (Boecker, Fischer, & Nieper, 1986) and
- design kits in Section 6 of this chapter.

4. The Need for Constrained Design Processes

Alan Kay (1984) 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 prevent 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 these 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 (Fischer, 1987);
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 reuse 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 (Fischer, 1987), 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 reuse 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 proceed. Therefore, real systems must be continuously *redesigned* (Fischer & Kintsch, 1986). *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 language ObjTalk) and our 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 (Fischer, 1987) 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 methods and 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.

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 be 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.

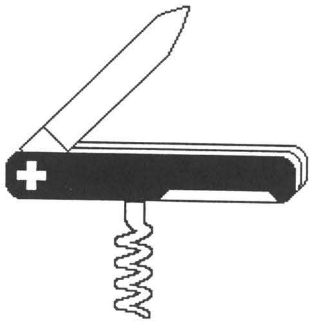
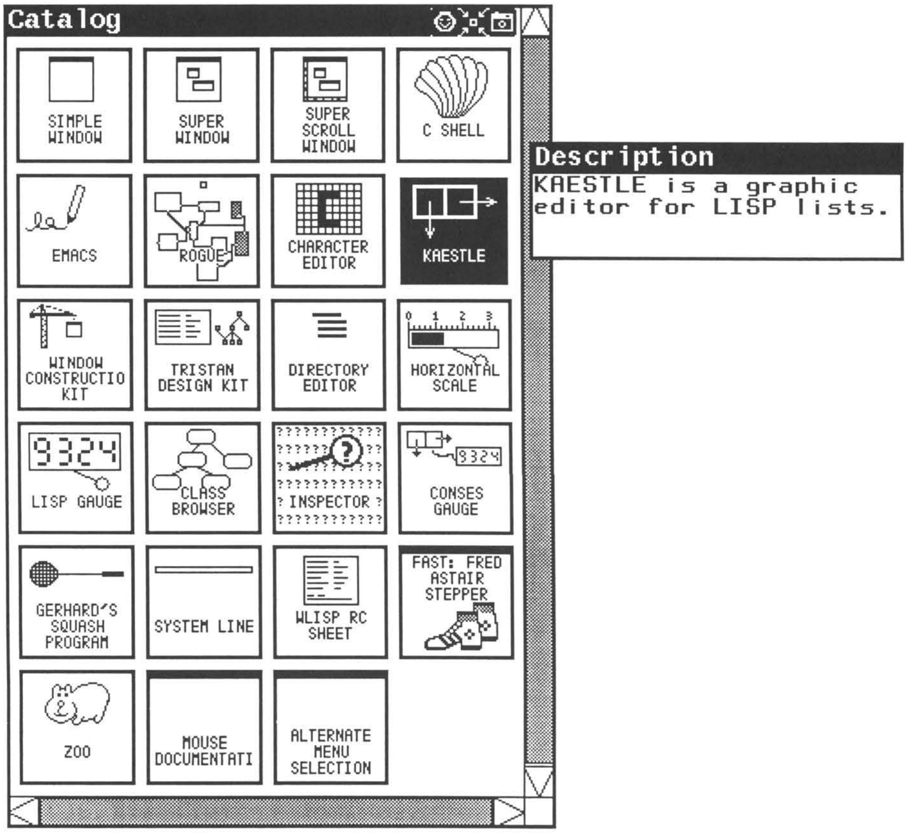


Figure 5-1: Selection of tools: the Swiss army knife



KAESTLE is a graphic editor for LISP lists.

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

Are selection type systems all that we need? We do not think so and agree with Alan Kay (1984) 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.*

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).

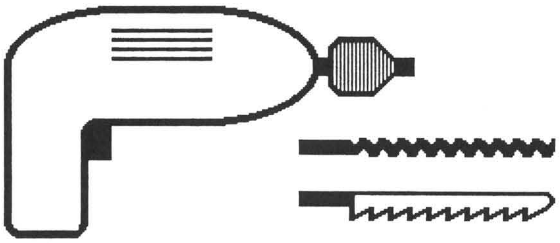


Figure 5-3: Simple combination of tools: the electric drill

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 Norman and Draper (1986)).

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. If enough classes exist, we can generate a broad range of behavior. In KBEmacs (Waters, 1985), instantiation is used to turn abstract cliches into program code.

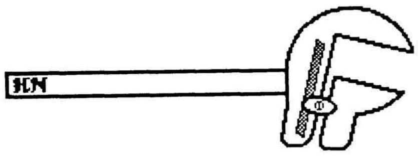


Figure 5-4: Instantiation: the adjustable end wrench

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 (Figures 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, 1983)), 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 on how to exploit it's power 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 which allows user modifiability should have an associated design kit. Design kits can contribute towards the achievement of the following goals: to resolve, at least partially, the basic design conflict of generality and power versus ease of use; to make the computer behind a system invisible; to allow the users to deal primarily with the abstractions of the problem domain (human problem-domain communication); and, finally, to protect the user from error messages and from 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 existing components (Fischer, Lemke, & Rathke, 1987).

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 for* u,v,...). Object-oriented formalisms (Lemke, 1985; Rathke, 1986) 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, 1985). Figure 6-1 shows an example screen. WLISP is an object-oriented user interface toolkit and programming environment implemented in FranzLisp and ObjTalk (Rathke, 1986), an object-oriented extension to Lisp.

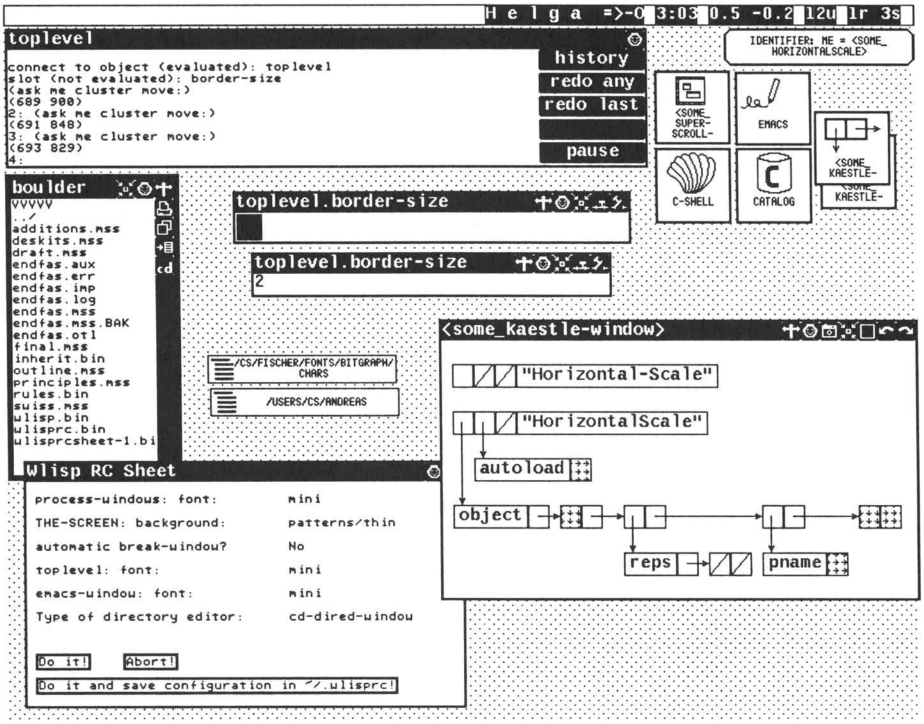
The object-oriented nature of the system provides a good framework to represent the entities of the toolkit (windows, menus, push buttons, etc.). Figure 6-2 shows an ObjTalk inheritance hierarchy of some simple window classes. In the following sections, we will see how design kits can help build new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



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

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

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

Figure 6-1: The WLISP programming environment

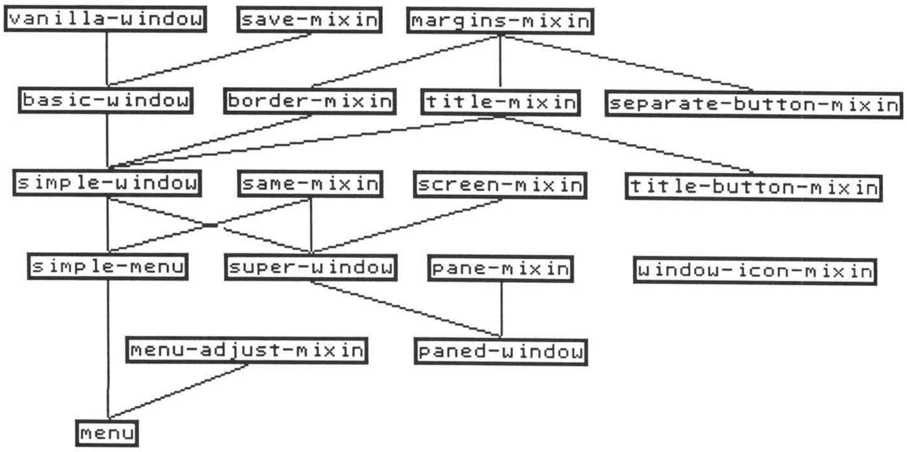
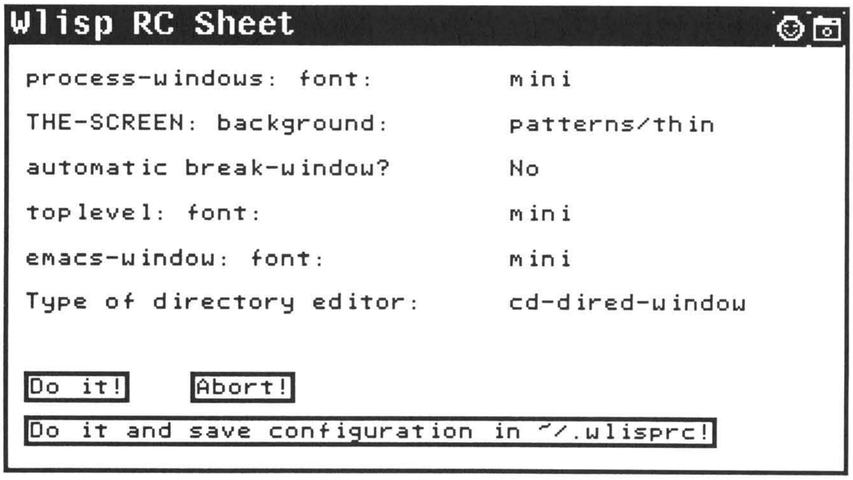


Figure 6-2: The inheritance hierarchy of windows

- 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).


Wlisp RC Sheet

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

Figure 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 cycle 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.

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 that can contain arbitrary program code provide the same, 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