From Interactive to Intelligent Systems

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

Abstract

Feature-rich software systems of today are the result of the continuous increase in computational power and growing requirements for broad functionality; these systems have to be mastered by casual or untrained users. This leads to operability problems (systems are too complicated), ineffective use, erroneous behavior and frustration. Careful empirical studies indicate that even in current systems only *a small percentage* of the available functionality is actually used. The availability of more computational power in the future will be of little value in constructing more usable systems, unless we open up new access paths to enable the user to take advantage of this increased functionality.

We claim that knowledge-based systems with qualitatively new human-computer communication capabilities are one of the most promising ways to create *intelligent systems*. We propose to extend the comprehensibility of systems by dedicating a large fraction of the computational power of the machine to sophisticated user support systems.

1. Introduction

The overall goal of this paper is to illustrate how to move on from interactive to intelligent systems. Two research areas are crucial for this transition: *knowledge-based systems* and *human-computer communication*. General principles for the design of intelligent systems will be postulated and a variety of system components will be described which we have designed and implemented over the last few years. The role of ergonomics research in relationship to intelligent ssytems will be discussed. Despite the fact that progress has been made towards the goal of making intelligent systems more a reality, many challenges remain which will be briefly described in the last section.

NATO ASI Series, Vol. F22
Software System Design Methods. Edited by J.K. Skwirzynski
© Springer-Verlag Berlin Heidelberg 1986

2. Why Do We Need Intelligent Systems?

The microelectronics revolution of the 1970s made computer systems cheaper and more compact, with a greatly increased range of capabilities. Computing moved directly into the workplace and the home to the fingertips of a large number of people. Much of this power is wasted, however, if users have difficulties in understanding and using the full potential of their new systems. Too much attention has been given in the past to technical aspects which have provided inadequate technical solutions to real world problems, have imposed unnecessary constraints on users and have been too rigid to respond to changing needs. More *intelligent* software is needed which has knowledge about the user, the tasks being carried out and the nature of the communication process.

The increased functionality of modern computer systems, driven by the many different tasks that a user wants to do, will lead to increased complexity. Empirical investigations show that on the average only a small fraction of the functionality of complex systems is used. Figure 2-1 summarizes empirical investigations and careful observations of persons using systems like UNIX, EMACS, SCRIBE, LISP etc. in our environment. It also describes different levels of system usage which typically can be found within many complex systems.

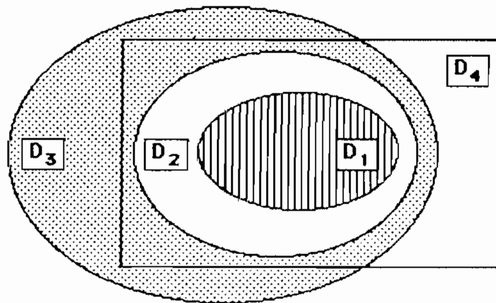

Figure 2-1: Different Levels of System Usage

The different domains correspond to the following:

D_1: The subset of concepts (and their associated commands) that the users know and use without any problems.

D_2: The subset of concepts which they use only occasionally. They do not know details about them and they are not too sure about their effects. Description of commands (e.g. in the form of property sheets), explanations, illustrations and safeguards (e.g. UNDOs) are important so that the user can gradually master this domain.

D_3: The mental model [Norman 82; Fischer 84] of the user, i.e. the set of concepts which she/he thinks exist in the system. A *passive help system* (see section 5.2.1) is necessary for the user to communicate her/his plans and intentions to the system.

D_4: Represents the actual system. Passive help systems are of little use for the subset of D_4 which is not contained in D_3, because the user does not know about the existence of these system features. *Active help systems* and *Critics* (see sections 5.2.2 and 5.3) which advise and guide a user similar to a knowledgeable colleague or assistant are required so that the user can incrementally extend her/his knowledge to cover D_4.

The only way to master complex systems is through incremental learning approaches. A partial knowledge of a system can lead to the following situation (see Figure 2-2): a user (based on her/his knowledge) makes a typing mistake which is interpreted (as an existing command) within the complete system and the user will be dumped in an area which is unfamiliar to her/him.

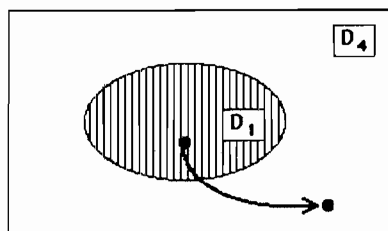

Figure 2-2: Protective Shields

Protective shields (based on the system's model of the user) are needed to avoid problems of this sort. These protective shields must not be so restrictive that they prohibit the exploration of additional system features by the user.

3. From Interactive to Intelligent Systems

Everyone today will agree that we do not want to operate computing systems in a batch mode any more. Interactive computing systems have made a major contribution to the widespread use of computers. Some of the most advanced interactive systems have been built as programming environments by Artificial Intelligence researchers [Barstow, Shrobe, Sandewall 84; Sheil 83] and around the SMALLTALK system [Goldberg 84].

3.1 The Architecture of Intelligent Systems

The next step will be to move on from interactive to intelligent systems. In intelligent systems a substantial part of the computational power will be used to document, explain and justify their expertise to others. They will provide insight, help and useful criticism so that a novice can slowly become an expert. They will augment human intelligence by providing visualization tools. Figure 3-1 describes our vision of the architecture of an intelligent system. It is important to note that the outer circle of system components in Figure 3-1 are not "add-ons" to existing systems, but should be an integral part of the system design right from the beginning.

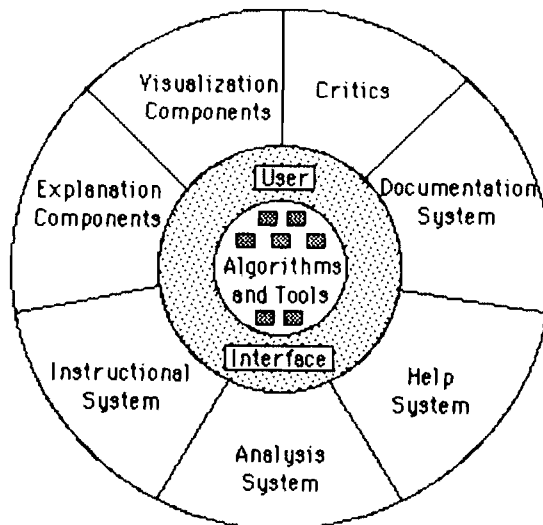

Figure 3-1: From Interactive to Intelligent Systems

Intelligent systems will contain many more tools than the functionally rich environments which are available to us today (e.g. the UNIX operating system, powerful AI programming environments). Empirical findings indicate that the following problems occur which prevent many people from successfully exploiting the potential of the systems available to us today:

- 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 tools produce for them,
- users cannot combine, adapt and modify a tool to their specific needs.

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

3.2 Human-Computer Communication (HCC)

Human communication and cooperation can serve as a model for how computing systems should be. What can humans do that most current computer systems cannot do? Human communication partners

- do not have the literalness of mind which implies that not all communication has to be explicit; they can supply and deduce additional information which was not explicitly mentioned and they can correct simple mistakes; empirical evidence shows that in any but the simplest human communication a substantial portion of the communicated message is not explicit;
- can apply their problem solving power to fill in details if we give statements of objectives in broad functional terms;
- can articulate their misunderstanding and the limitations of their knowledge;
- can provide explanations to others of how they reached a conclusion or why they behaved in a particular way;
- can solve problems by taking imaginative leaps, for example by conceiving of an analogous situation of similar characteristics with which they are more familiar.

Traditionally the relationship between the user and the computer was sufficiently remote that it was more like a literary correspondence than a conversational dialogue. Today the user is coupled directly with the computer which causes the following changes:

- interaction with computers is emerging as a human activity;
- the prior styles of interaction have been all extremely restricted (e.g. comparable to the driver of an automobile or the secretary using a typewriter); there was a limited range of tasks to be accomplished and a narrow range of means (e.g. like having a control stick in video games); the user was just an operator;
- the increased availability, the decreased cost and the greatly improved hardware allow that an increasing amount of computational resources can be spent on human-computer communication, rather than on purely computational tasks (see Figure 3-1);
- system designers tend to concentrate on the commands of the computer systems (just look at the documentation for almost any system, which is usually a catalogue of commands); yet it is how, when and why the commands are used that is most important to the user;
- there is a growing understanding that guidelines and classifications of the appropriate ergonomic dimensions for human-computer communication are less clear cut. Detailed prescriptions or check-lists cannot be provided to cover all aspects of human-computer communication because so much is dependent on human *cognitive* abilities - how people behave, think and perceive the world. Such subjective factors cannot be measured and predicted with the same precision that is possible with elements in a physical environment (see chapter 6).

Human-computer communication is more than drawing nice pictures on the screen. The viewers on the display screen are important, but they are of little use if there are no interesting knowledge structures behind them.

3.3 Knowledge-based Systems (KBS)

Knowledge-based systems are the most promising approach to qualitatively improve human-computer communication. Based on the above analysis of communication processes among humans the model in Figure 3-2 is suited to fulfill the stated requirements. It contains a knowledge base which can be accessed by both communication partners; this implies that the necessity to exchange all information explicitly between user and system does not exist any more.

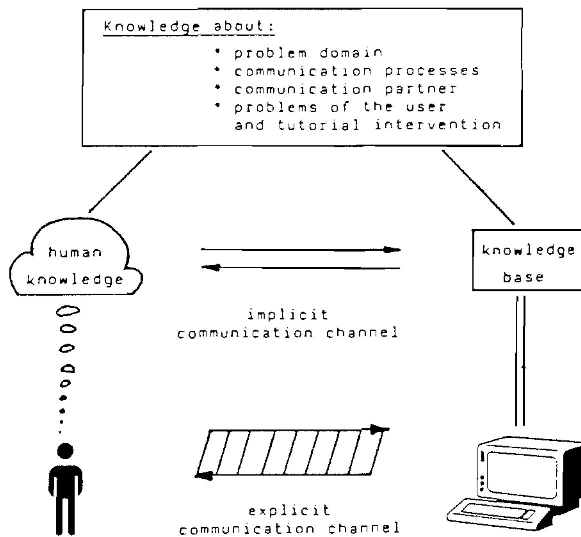

Figure 3-2: Architecture for Knowledge-Based Human-Computer Communication

This system architecture has the following advantages compared to current architectures:

1. the *explicit communication channel* is widened. The interfaces use windows with associated menus, pointing devices, color and iconic representations; the screen is used as a design space which can be manipulated directly.

2. information can be exchanged over the *implicit communication channel*. Both communication partners have knowledge which eliminates the necessity that all information has to be exchanged explicitly.

The four domains of knowledge shown in Figure 3-2 have the following relevance:

1. **Knowledge of the problem domain:** Intelligent behavior builds upon large amounts of

knowledge about specific domains. This knowledge imposes constraints on the number of possible actions and describes reasonable goals and operations. If, for example, in UNIX a user needs more disk space it is in general not adequate help to advise him to use the command "**rm ***" (the command will delete all files in the current directory; [Wilensky et al. 84]) although it would perfectly serve her/his explicitly stated goal. The user's goals and intentions can be inferred if we understand the correspondence between the system's primitive operations and the concepts of the task domain.

2. **Knowledge about communication processes:** The information structures which control the communication should be made explicit, so the user can manipulate them.

3. **Knowledge about the communication partner:** *The* user of a system does not exist; there are many different kinds of users, and the requirements of an individual user grow with experience. To pay attention to individual differences the following knowledge structures have to be represented:
   - The user's conceptual understanding of a system (e.g. in an editor, text may be represented as a sequence of characters separated by linefeeds which implies that a linefeed can be inserted and deleted like any other character).
   - The user's individual set of tasks for which she/he uses the system (a text editor may be used for such different tasks as writing books and preparing command scripts for an operating system).
   - The user's way of accomplishing domain specific tasks (e.g. does she/he take full advantage of the systems functionality?).
   - The pieces of advice given and whether the user remembered and accepted them.
   - The situations in which the user asked for help.

   A prerequisite for knowledge-based human-computer communication is to monitor the user's behavior and reason about her/his goals. Sources for this information are: the user's actions including illegal operations. This is based on the hypothesis that *a user does not make arbitrary errors; all operations are iterations towards a goal* [Norman 82].

4. **Knowledge about the most common problems which users have in using a system and about instructional strategies:** This kind of knowledge is required if someone wants to be a good coach or teacher and not only an expert; a user support system should know when to interrupt a user. It must incorporate instructional strategies which are based on pedagogical theories, exploiting the knowledge contained in the system's model of the user. Strategies embodied in our systems [Fischer 81] include:
   - *Take the initiative* when weaknesses of the user become obvious. Not every recognized suboptimal action should lead to an intervention.
   - *Be non-intrusive.* Only frequent suboptimal behavior without the user being aware of it should trigger an action of the system.
   - *Give additional information* which was not explicitly asked for but which is likely to be needed in the near future.
   - Assist the user in the *stepwise extension* of her/his view of the system. Be sure that basic concepts are well understood. Don't introduce too many new features at once.

The main issues in building knowledge-based systems (which are actively pursued in Artificial Intelligence research) are:

1. **knowledge acquisition:** how is knowledge acquired most efficiently from human experts

and from data gathered by instruments? Can the experts themselves directly manipulate the knowledge base or do they need a knowledge engineer?

2. **knowledge representation:** how can the needed knowledge for complex problem solving processes be represented to be effective for the inference engine and to be understandable for the human?

3. **knowledge utilization:** how can we retrieve the relevant knowledge needed in specific situations? Does the knowledge base help us in finding the relevant knowledge? Does it support browsing techniques to navigate through a knowledge space whose structure and content is unknown to the user in advance?

We will briefly describe our work on knowledge representation which has led to the development of *ObjTalk* [Laubsch, Rathke 83; Rathke, Lemke 85; Lemke 85], an object-oriented knowledge representation language and programming environment in which most of our software is implemented.

3.4 Object-Oriented Knowledge Representation

A major strength of object-oriented knowledge representations is their ability to provide the designer for many problems with a concise and and intuitively appealing means of expression. The claim of intuitive appeal is based on our experience that object-oriented styles of description often closely match our understanding of the domain being modeled and therefore lessens the burden of reformulation in developing and understanding a formal description. The implementation of our window, menu and icon systems [Boecker, Lemke, Fabian 85] serves as a convincing example for this claim.

ObjTalk is an object-oriented programming language as well as a general formalism for knowledge representation. *ObjTalk* as a programming environment includes the following features:

- *message-passing* as the basic model for computation;
- *class-instance* relationship; instances of a class are specified by filling the "slots" described in their class with specific values; their class also provides a description of the messages that they understand and how to respond to them; these descriptions are called *methods*;
- *class-inheritance* framework (with multiple superclasses); it supports the methodology of "programming by specialization"; the *method* of a class may be extended or replaced by a *method* with the same name contained in one of the subclasses of the class.

There are a variety of features that make *ObjTalk* an interesting formalism for knowledge representation; among them are the following:

- *if-set, if-changed, if-needed* and *if-forget* demons (which are triggered automatically when the corresponding operations are carried out);
- Rules (as in rule-based expert systems) which may be associated with classes; they may be used to automatically update relationships among the slots of (an instance of) a class.

As mentioned above, a major application of *ObjTalk* is our window system (see for example Figures 5-3 and 5-4 in section 5.4 for two applications using the window system and Figure 3-3 for the inheritance network underlying the window system).

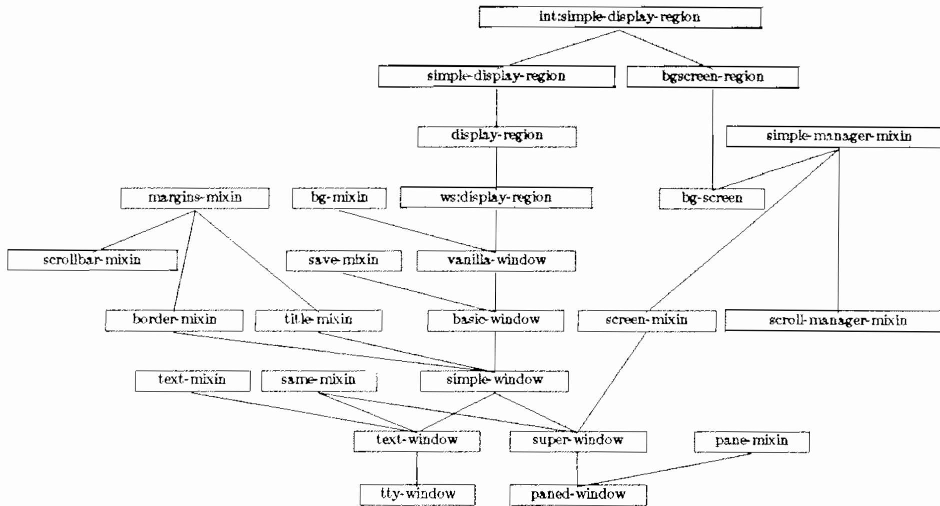

Figure 3-3: The *ObjTalk* Inheritance Hierarchy of our Window System

The facilities of *ObjTalk* allowed for the highly modular and flexible structure of the window system and they led to an open architecture providing high level interfaces to applications and at the same time keeping lower level parts of the system accessible for situations where nonstandard extensions are needed.

4. Design Considerations for Intelligent Systems

In this chapter we summarize major principles that have emerged from our work. They are based on a growing understanding of human information processing capabilities and on our prescriptive goals about system design. These principles provide a basis for developing and assessing some aspects of intelligent systems.

4.1 Design Considerations Based on the Human Information Processing System

Users of computer systems are humans and the human information processing system has strengths and weaknesses. Examples of the human mental processing limits are the limited short term memory and our tendency to make errors (for a great variety of reasons). Examples of the strength is the great power of our visual system which is slowly taken more into account in the new two-dimensional interfaces, using iconic and spatial information as well as color.

We postulate a few principles which provide the most important design criteria derived from the human information processing system:

1. *The limited resource in human information processing is human attention and comprehension, not the quantity of information available.* Modern communication and information technologies have dramatically increased the amount of information available to individuals. Most persons will have access to more information than they can deal with. An important function of future computing systems is to allow for the selection of the information we actually want and need and presentation of it in the most appropriate way.

2. *In complex situations, the search for an optimal solution is a waste of time.* There are limits to the extent to which people can apply rational analyses and judgements to solving complex, unpredictable problems. It is insufficient to ask people to "Think more clearly" without providing new tools such as knowledge-based systems, which help extend the boundaries of human rationality. The aim is to achieve the most satisfactory solution (i.e. we have to "satisfice" instead of "optimize", see [Simon 81]) given current knowledge, accepting that better solutions will emerge as a result of experience and enhanced knowledge and understanding.

3. *The nature of human memory mechanisms are important design considerations.* The limitations and structure of human memory must be taken into account in system design. People have limited short term memories. Dialogues should, therefore, be constructed which do not expect the user to remember everything and which reinforce, prompt and remind the user of necessary information in a supportive but unobtrusive way. Recognition memory and recall memory are two memory structures with different access mechanisms which are relevant to judge the advantages and limitations of function-key versus menu-based systems.

4. *The efficient visual processing capabilities of people must be utilized fully.* New technologies (like raster displays) have opened ways to exploit human visual perception more fully, e.g. through the use of windows, color, graphics, icons and mice. These technologies support an interaction technique like "direct manipulation" which is an important alternative in human-computer communication to be further explored in future systems.

5. *The structure of a computer system must be understandable by people using it rather than requiring the user to learn by rote the functions that can be performed.* An adequate understanding of how a system works gives users the knowledge and confidence to explore the full potential of a system, which can have a vast range of different options. Learning by rote may train the user to operate a limited number of functions but makes it difficult for the user to cope with unexpected occurrences and inhibits their exploration of the full potential of the system.

4.2 Design Considerations Based on our Prescriptive Goals for Intelligent Systems

Our systems have to be flexible enough to support the variety of behavior occasioned by the unpredictable details of particular situations. The design of hard- and software and human-computer communication capabilities must be responsive to the knowledge worker's amorphous responsibilities. We must base our theories, methodologies and tools on an understanding of what users are doing. The work of the users we want to support can be characterized along the following dimensions:

- to deal with fuzzy problems, with instabilities in specifications and uncertainties; the existence of partial solutions incrementally increases the user's understanding of the problems to be solved and allows her/him to analyze a prototypical situation instead of anticipating it;
- formal approaches (e.g. methods from operations research) fail in most cases, because we do not understand the problems well enough;
- the space of possibilities is unlimited; choosing is not always good enough, in many situations it is necessary to generate new solutions; this requires the exploration of unknown situations and not the application of habitual means;
- in complex decision making there are always too many things which are not articulated; therefore in many situations, tasks can not be delegated because they cannot be described well enough for an assistant to do them.

Based on this brief description we postulate additional principles which can serve as further guidelines to design intelligent systems:

1. *There is no such thing as the user of a system; there are many different kind of users and the requirements of an individual user grows with experience.* Computer systems must adaptively grow with the experience of the user. The heterogeneity of the user community and the growing experience of one user working with a system over a long period of time requires an adaptive behavior based on a system model of the user's abilities for a specific task.

2. *The "intelligence" of a complex tool must contribute to its easy use.* Truly intelligent and knowledgeable human communication partners, 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 system should be used to provide effective communication.

3. *The user interface in a computer system is more than just an additional component; it is an integral and important part of the whole system.* The traditional design process, proceeding from the "inside" to the "outside", has to be reversed as much as possible. The design, development and evaluation of new information systems should start with an understanding of the overall social and technical environment in which any particular new technology is embedded (see Figure 4-1).

5. Components of Intelligent Systems

Over the last several years we have designed and implemented the following prototypical components of intelligent systems (see Figure 3-1):

- documentation systems,
- passive and active help systems,

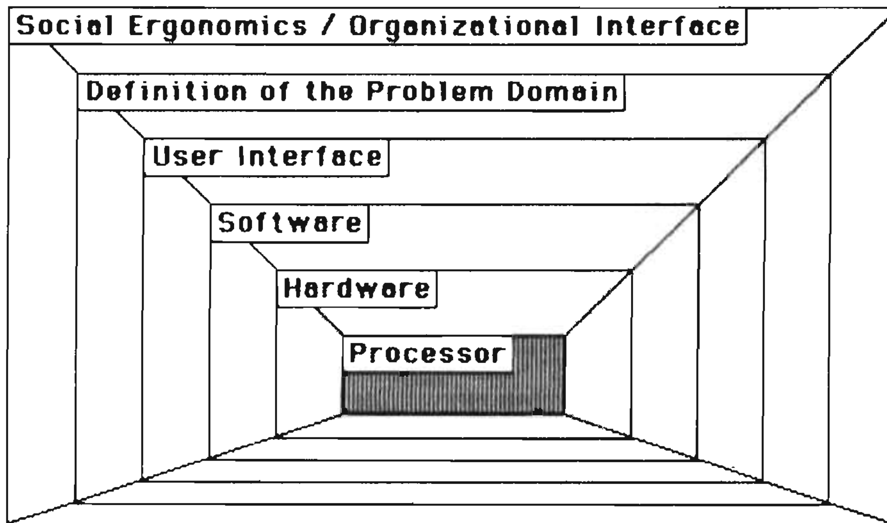

Figure 4-1: Different Levels in a System Development Process

- critics,
- visualization tools.

These components will be briefly described in the next sections. Most of them are of equal importance to the system designer and the system user.

5.1 Documentation Systems

A program documentation system should be a part of a software engineering environment. Its importance stems from the large range of different tasks documentation is used for:

- to serve as a communication medium between different people (clients, designers and users),
- to enhance the designers understanding of the problem to be solved and to assist them in improving the problem specifications,
- to support the designers during the implementation of their solution,
- to enable programmers to reuse a program and to extend existing systems to tool kits,
- to maintain a programming system,
- to make a program portable and sharable in a larger community.

In our design and implementation [Lemke, Schwab 83; Fischer, Schneider 84] a program documentation system is a knowledge base containing all of the available knowledge about a system combined with a set of tools useful for *acquiring, storing, maintaining* and *using* this knowledge.

The knowledge base is

- in part interpreted by the computer to maintain the consistency of the acquired knowledge about structural properties; it supports the users in debugging and maintaining their programs;
- in part only useful for the user, i.e. not directly interpretable by the machine. In this case the machine serves as a medium for structured communication between the different users. The computer can support the user to maintain the non-interpretable information in the knowledge base and do user-guided changes of information.

A documentation system should support the entire design and programming process. Valid and consistent documentation is of crucial importance during the programming process itself. The information structures that are accumulated around a program can be used to drive an evolutionary and incremental design process.

In the traditional view documentation is created at the end of the programming process; in our model documentation serves as the communication medium for all people involved with a software product. Documentation is useful throughout the entire process and serves as a starting point for new solutions of the problem. The purpose of documentation in this view is comparable to that of a proof in mathematics: a crystallization point for new ideas.

We gain the full benefit of a program documentation system only, if it is an *integral* part of a programming environment. Program documentation produced as a separate document by a word processing system has the following disadvantages:

- it is impossible to provide pieces of information automatically,
- it is impossible to maintain consistency between the program and its documentation automatically (or at least semi-automatically),
- it is impossible to generate different external views dynamically from one complex internal structure (e.g. to read a documentation either as a primer or as a reference manual),
- it is impossible to create links between the static description and the dynamic behavior.

Program documentation for whom? Program documentation has to serve different groups who try to perform different tasks. Therefore the amount and quality of information offered to these groups of people has to be different. We distinguish the following groups and their tasks:

- the *designers* of a system during the programming process. They have to have access to their design decisions and the different versions of the system. They also need information about the state of their work during the whole design process.
- the *programmers* who are trying to reuse or modify a program that they do not know yet. They want to understand the purpose and algorithms of the program to decide which parts of it have to be changed to fit their needs. They need information about design decisions (in order to avoid known pitfalls) as well as a thorough documentation of the existing code.
- the *clients* who are trying to find out whether the implemented system solves their problem. They want to improve their own understanding by working with a prototypical version of the system and are therefore not interested in any programming details but in design decisions.
- the *users* want to see a description in terms of "What does it do? How can I achieve my goals?"; for end-users the documentation has to offer different views of the system: a primer-like description for the beginner and manual-type explanations for the expert.

Knowledge acquisition and updating. The information structures which are used in our system come from two sources:

- a program analysis system provides information about the structural properties (cross references, side effects) of a program. The users do not have to provide information that can be created automatically, so they are free to concentrate on the creative aspects of their work.
- the programmers have to provide semantic information about the different parts of the program, information about the internal (semantic) structure of their systems, descriptions of the algorithms used etc..

Most of the analysis done by the system is done at read-time. This means that we have to do the analysis after each alternation of the program code. The system knows about possible dependencies between knowledge units and, if necessary, reanalyzes the units in question. It informs the programmer about possible inconsistencies in the knowledge base. These techniques help us to maintain the consistency between different representations of the information.

The way the system decides if a knowledge unit has to be updated is the following:

- the system knows that it has to change certain structural information (e.g. "calls" and "is-called-by" relations) automatically. The system is able to alter information by using its cross-reference knowledge. This knowledge can also be used to guide the users to places where they possibly want to change information.
- for each unit users can provide a list of other knowledge units they want to inspect and possibly alter if a unit has been updated.

Using the available knowledge. A knowledge-based program documentation system is only useful if the relevant information can be easily obtained. The following two requirements must be supported:

- *availability:* the knowledge about the system (incorporating the consequences of all changes) must be available at any time.
- *views of reduced complexity:* the structures in our knowledge base are too complex to be used directly. A filter mechanism where the filters can be defined by the user [Lemke, Schwab 83] allows the generation of views of reduced complexity showing only the information which is relevant for a specific user at a specific time.

5.2 Help Systems

The following system descriptions [Fischer, Lemke, Schwab 84; Fischer, Lemke, Schwab 85] are based on preliminary investigations for a passive and an active help system for a screen-oriented editor. *Passive* help systems are needed if the users know their goals but cannot communicate them to the system. *Active* help systems assist users when they are not aware that the system offers better ways to achieve a task (see Figure 2-1).

5.2.1 *Passivist*: A Passive Help System

The first step in the design of *Passivist*, a natural language based help system, was to get an idea of the real needs of the user. In several informal experiments a human expert simulated the help system in editing sessions with users of different expertise. The results indicated a fairly diverse set of problems ranging from finding keys on the keyboard to complex formatting tasks.

Passivist provides help to requests such as:

- *How can I get to the end of the line?*
- *I want to delete the next word.*

Passivist uses a help strategy in which each step of the solution is presented and explained to the user who then executes this step and immediately sees the resulting effects. Help is given as text generated from sentence patterns according to the goal structure of the problem solving process and key sequences and subgoals are displayed graphically.

Passivist is implemented in OPS5 [Forgy 81]. Flexible parsing using OPS5 is achieved by a rule-based bottom-up method. The consistent structure of the system as a set of productions and a common working memory allows the use of the same knowledge in several stages of the solution process. For example, knowledge about the state of the editor is not only used to select a possible solution for the user's problem but also to disambiguate the user's utterance. In the phrase *the last line* with the cursor being at the beginning of the editing buffer it is clear that the user refers to the last line of the buffer (and not to the previous one).

The following rule (an English-like transcription of the corresponding OPS5 rule) represents the systems knowledge about deleting the end of a line:

```
IF   the goal is to delete a string
         AND  the string is the end of the current line
         AND  the cursor is not at the end of the current line

THEN remove the goal from the working memory
         AND  create a new goal to propose the command "rubout-line-right"
```

5.2.2 *Activist*: An Active Help System

Activist, an active help system for a screen-oriented editor, is implemented in *FranzLisp* and *ObjTalk* (see section 3.4).

Activist deals with two different kinds of suboptimal behavior:

1. the user does not know a complex command and uses suboptimal commands to reach a goal (e.g. she/he deletes a string character by character instead of word by word).
2. the user knows the complex command but does not use the minimal key sequence to issue the command (e.g. she/he types the command name instead of hitting the corresponding function key).

Similar to a human observer, *Activist* handles the following tasks (for details see [Fischer, Lemke, Schwab 84] and [Fischer, Lemke, Schwab 85]):

- to *recognize* what the user is doing or wants to do,
- to *evaluate* how the user tries to achieve her/his goal,
- to *construct a model of the user* based on the results of the evaluation task,
- to decide (dependent on the information in the model) *when* and *how* to interrupt (tutorial intervention).

In *Activist* the recognition and evaluation task is delegated to 20 different *plan specialists*. Each one recognizes and evaluates one possible plan of the problem domain. Such plans are for example *"deletion of the next word"*, *"position at the end of line"*, etc..

A plan specialist consists of:

- A transition network (TN), which matches all the different ways to achieve the plan using the functionality of the editor. Each TN in the system is independent. The results of a match are the *used editor commands* and the *used keys* to trigger these commands.
- An expert which knows the optimal plan including the *best editor commands* and the *minimal key sequence* for these commands.

Figure 5-1 displays the user model that *Activist* has built up. For each plan there is a pane which shows the performance of a specific user concerning this plan. Panes with black background indicate that the corresponding plan is currently not monitored by the active help system.

give COMMAND: set-cursor-to-beginning-of-line

(set-cursor-to-beginning-of-line) liegt auch auf ^A

Figure 5-1: The User Model of *Activist*

The dialogue window at the bottom displays a help message given to the user. She/he has executed the command *set-cursor-to-beginning-of-line* by typing in the command name. *Activist* gives the hint, that this command is also bound to the key *CTRL-A*.

Figure 5-2 relates to the monitoring of the plan *"delete-the-left-part-of-the-current-word"*. The window shows one pane of Figure 5-1 in more detail: the proposed command (with the optimal keys) for this plan and the state of the plan recognition.

The user has executed the command *rubout-character-left* with the DEL-key three times. After these actions the cursor is located to the right of the first character of the word. If the user once again invokes *rubout-character-left* the plan will be recognized. Then the evaluation will begin: the commands used will be compared with the optimal commands for this plan and *Activist* will recognize the first kind of suboptimal behavior (as described above). Based on the instructional strategies chosen *Activist* may then decide to interrupt the user and describe how to use the *"delete-the-left-part-of-the-current-word"* command.

```
DELETE left part of word
U S E R   M O D E L

plan executed:                  1
good done:                      0
wrong command used:             1
with unnescessary keys:         6
command with wrong keys used:   0
with unnescessary keys:         0
messages sent to user:          0

I N T E R N A L   I N F O R M A T I O N

proposed commands:  rubout-word-left
optimal keys:       ESC h

commands: rubout-character-left rubout-character-left rubout-character-left
keys:( DEL )( DEL )( DEL )
automaton in state: Start
```

Figure 5-2: Monitoring the Recognition Task

5.3 Critics

In our current work we extend the work on active help systems in order to develop systems which can serve as *critics*. This is done primarily by representing more elaborate knowledge structures in the computer. The following four information structures are prerequisites to support systems which can serve as *Critics*:

- *Characteristics of the User:* General knowledge of the user's abilities in the subject domain independent of the current problem. This knowledge can be viewed as an abstraction from individual problem solving attempts.
- *The User's Problem Solving Approach:* The system's idea of the problem solving path chosen by the user. This may be an explicit hierarchical model of the design choices which led to the code produced by the user where the user explicitly indicates in which way she/he wants to

attack the problem. Plan recognition strategies attempt to infer the user's plan through observation of her/his steps performed in solving a task. For these systems the plan essentially *is* the user model.

- *Task Model:* The system's understanding of which problem the user currently wants to solve. In tutorial systems this knowledge can be built into a system in advance. In systems which can serve as critics (i.e. they have to support the users in their own doing) we need methods to infer the current task.
- *Domain Knowledge:* User independent expert knowledge of the selected domain of competence of the system.

Given this detailed model of the user and the task domain, the following actions of the system become possible (e.g. to support learning strategies like *learning on demand*):

- *Select appropriate actions with respect to the user:* If the comparison of the user model and the system's expert knowledge reveals weaknesses in a specific area, the system should only become active if this area is adjacent to already known areas and does not require too many other areas unknown to the user.
- *Select examples from the domain the user is familiar with:* By using an executable form of representation it is possible to generate illustrations out of areas which the user already understands and thus reduce the cognitive distance that has to be bridged.
- *Present only the missing pieces of knowledge:* In dialogues a large amount of time is spent to find out what each communication partner knows and does not know about the subject area. With a detailed user model, the system can concentrate on the very points where the user needs help.
- *Better understanding of the user:* Using knowledge about the user's understanding of a problem domain makes it much easier to find out about her/his real problem. We encountered many cases where a user had a problem which originated in a wrong decomposition of a higher level problem. Using knowledge about the user it is possible to trace a problem back to its real roots.

The Code-Improver System. Our currently existing system, called *Code-Improver,* is used to get ideas on how to improve *Lisp* code. Improvements may be in either of two directions (which can be chosen by the user):

- to make the code more *cognitively* efficient (e.g. more readable or concise) or
- to make the code more *machine* efficient (e.g. smaller or faster); this improvements include those that can be found in optimizing compilers.

The system is in regular use by different groups for slightly different reasons:

- by intermediates who want to learn how to produce better *Lisp* code; we are currently testing the usefulness of the tool by gathering empirical, statistical data using advanced undergraduate students of an introductory *Lisp*-course as subjects;
- by experienced users who want their code to be "straightened out"; instead of doing that by hand (which these users in principle would be able to), they use the system to carefully reconsider the code they have written. The system is used to detect optimizations and simplifications that can be done to the code. The system has proven especially useful with code that is incrementally developed, i.e. gets changed and modified continuously.

The system operates by using a set of (> 200) transformation rules that describe how to transform 'bad'

code into better one. The user's code is matched against these rules and the transformations suggested by the rules are given to the user; the code is not modified automatically. It is important to note, that the system is *not* restricted to a specific class of *Lisp* functions or application domain. It accepts whatever *Lisp*-code is given to it. However, there is a trade-off: since the system does not have any knowledge of specific application areas or algorithms it is naturally limited in the kind of improvements that derive from its more general knowledge about programming. The improvements suggested by the system are of the following kind:

- suggesting the use of macros (e.g. `(setq a (cons b a))` may be replaced by `(push b a)`);
- replacing compound calls of *Lisp* functions by simple calls to more powerful functions (e.g. `(not (evenp a))` may be replaced by `(oddp a)`);
- specializing functions (e.g. replacing `equal` by `eq`);
- using integer arithmetic wherever possible;
- finding alternative (simpler or faster) forms of conditional or arithmetic expressions;
- eliminating common subexpressions;
- replacing 'garbage' generating expressions by non-copying expressions (e.g. `(append (explode word) chars)` may be replaced by `(nconc (explode word) chars)`);
- finding and eliminating 'dead' code (as in `(cond (...) (t ...) (dead code))`);
- (partial) evaluation of expressions (e.g. `(sum a 3 b 4)` may be simplified to `(sum a b 7)`).

The current version of the *Code-Improver* system runs in batch mode. Like the "writers-workbench" UNIX tools, *diction* and *explain*, it is given a file containing *Lisp* code and comes back with suggestions on how to "improve" that code.

The problem of knowledge acquisition for the system's model of the user is solved using program code written by the user. Our current techniques will be extended from recognizing pieces of code which can be improved to both recognizing the goals of a piece of code as well as existing and missing concepts which led to its generation. Since only in very few cases a definitive assumption about the knowledge of the user can be made, it is important to have many clues which allow to make uncertain inferences when no specific evidence is available.

5.4 Visualization Tools

Many communication problems between humans and computers are due to the fact that the strongest information processing subsystem of the human brain, the *visual system*, is hardly utilized by current software systems. Most systems present the result of computational processes in textual and symbolic ways that are not suited to the human information processing capabilities. Over the last several years we have actively pursued the goal of constructing a *software oscilloscope* whose visualization techniques enhance the communication process (see [Boecker, Nieper 85; Boecker, Fischer, Nieper 85]) by broadening the explicit communication channel between computer and machine (see Figure 3-2).

5.4.1 Visualization of Data Structures: The *Kaestle* System

The most important data structure of *Lisp* is the *list*. With *Kaestle* [Boecker, Nieper 85] the graphic representation of a list structure is generated automatically and can be edited directly with a pointing device. By editing we do not only mean changing the structure itself but changing the graphic representation, the layout, of the structure. *Kaestle* is integrated into a window system and multiple *Kaestle*-windows may be used at the same time. The *user interface* is menu-based (see Figure 5-3) and the *program interface* is realized through *ObjTalk* methods which can be triggered by sending messages to a *Kaestle*-window.

Kaestle provides the following *functionality*:

1. generating a graphic display (multiple independent structures may be displayed at a desired position);
2. changing the graphic representation (additional display of structures which are truncated in the current display, deleting parts of the structure from the screen and moving parts of the structure on the screen);
3. changing the structure (the manipulation (insertion of atoms or pointers) of the graphic representation immediately changes the underlying structure);
4. controlling the layout planning in advance (selecting the maximum depth and length of lists to be displayed, selecting the maximum length of atom names to be displayed, selecting a planning algorithm (car-first or cdr-first algorithm) and selecting the font to be used);
5. general undo and redo mechanisms.

The user of *Kaestle* can take one of the following roles:

1. an *active* role: A graphic representation can be generated from whatever the user types in and the user is encouraged to an exploratory style of learning.
2. a *passive* role: An inexperienced user does not know which structures and which operations on them lead to interesting effects. To display prestored examples or examples taken from the actual context, *Kaestle* can be used through a program interface, i.e. programs can be written which generate graphic representations in a movie-like manner.

We will combine *Kaestle* with the documentation, help and critic tools described in the previous sections. *Kaestle* is a tool which will be used by the *Code-Improver* to illustrate explanations given by the system to answer questions like:

- What is the difference between several list creation functions (e.g. `cons` and `list`)?
- What is the difference between `equal` and `eq`?
- What is the difference between non-destructive and destructive functions (e.g. `append` and `nconc`)?
- Why is it possible to transform `(append (explode word) chars)` to `(nconc (explode word) chars)`?
- Why is it wrong to transform `(append chars (explode word))` to `(nconc chars (explode word))`?
- How is a stack implemented in *Lisp*? What are `push` and `pop` doing?

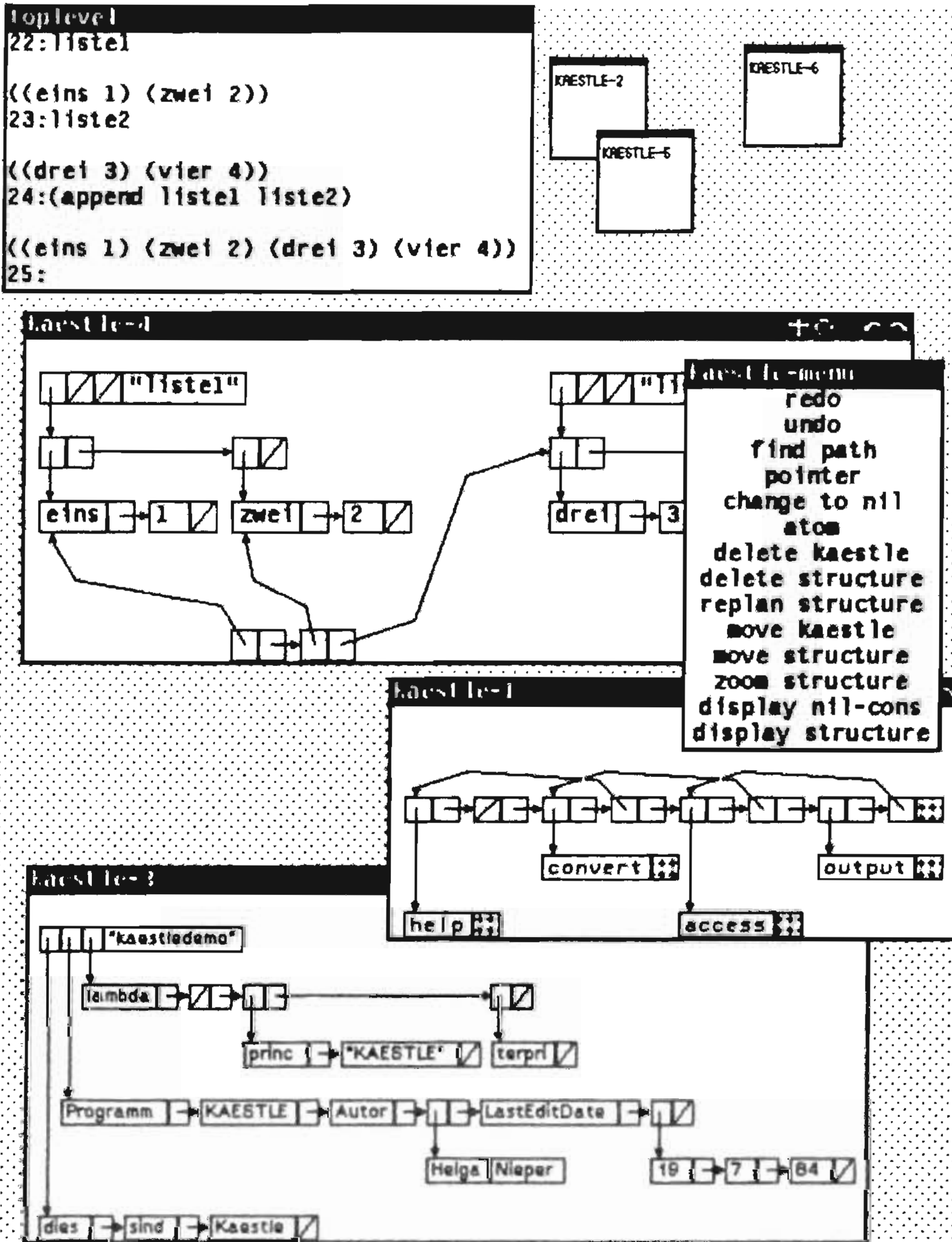

Figure 5-3: *Kaestle*: Visualization of Data Structures

5.4.2 Visualization of Control Structures: The *FooScape* System

One of the most helpful tools for understanding programs which are composed of a large set of procedures is a *program tree* that displays the hierarchical calling structure (see Figure 5-4, "Dynamic Calling Tree"). In reality however, most programs are more appropriately described as a *network of functions* rather than a tree (see Figure 5-4, "FooScape"). In *FooScape* functions are displayed as ellipses and an arrow connects ellipses *a* and *b* if function *a* calls function *b*.

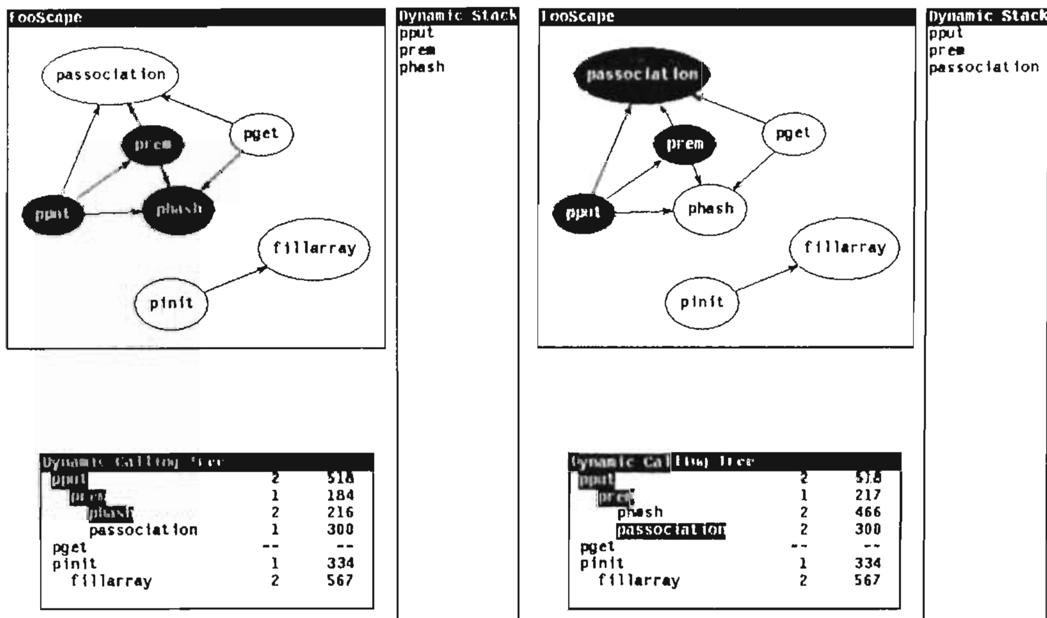

Figure 5-4: *FooScape*: Showing the Dynamic Behavior of a Program

The planning of the layout (placement of the ellipses and arrows) is done automatically. However, because the "beauty" of the solution sometimes is not acceptable we allow the user to modify the layout interactively by moving "bubbles" around or by altering the set of functions included in the display. The interaction style is similar to the one of the *Kaestle* system.

FooScape can be used to display the dynamic behavior of programs. Figure 5-4 shows two snapshots. A function name is highlighted - i.e. flips from white to black - whenever the function is active. *FooScape* can be augmented by a tool that displays the current stack of function calls. Traditional techniques for monitoring the *dynamic behavior* of programs (eg. breakpoints, dumps) capture just one state of the data and too often generate a huge amount of data. The animated *FooScape* system tries to avoid these disadvantages, while, on the other hand, trying to preserve the dynamics of the processes being monitored.

6. The Role of Ergonomics Research in Intelligent Systems

Ergonomics analyzes the consequences of technology to the learning and working environment of a human. It is dominated by the goal to create environments which support the human and do not require that the human has to put up with inadequate technical solutions.

In the past properties of systems were investigated which could be measured with methods from physics (e.g. according to the size of an average hand, what is the optimal distance between the keys on a keyboard). This approach is insufficient for the design and evaluation of human-computer communication and knowledge-based systems.

Design conflicts. The first thing a designer is confronted with is that *there are no optimal solutions -- only tradeoffs*. One has to cope with conflicting design issues; some of the major ones are:

- cognitive efficiency versus machine efficiency of a system,
- simplicity of a tool (to be easily handled by a large community) versus power (to be used in the construction of a large variety of different and complex systems),
- the necessity to remain compatible with existing systems (e.g. timesharing computers) versus exploiting the power of new media (e.g. networks of personal machines),
- ease for novice users (like mnemonic names in editors or menu-driven systems) versus convenience for experienced users (like a terse command language),
- tight integration between different subsystems versus flexible, reconfigurable modules or tool kits.

Costs of ignoring ergonomics. Without a great concern for ergonomics (especially software ergonomics) computing systems may prove detrimental to the people who have to work with them. Once a computer system has been installed, it is difficult to avoid the assumption that the things it can deal with are the most relevant things (e.g. text-processing systems do not support graphics, therefore papers get written without graphics). Introducing a system whose functioning is incompletely understood may cause unintended transfers of power and obscured responsibilities. Designs which ignore the fact that computing systems are embedded systems (see Figure 4-1) may lead to systems where satisfying work procedures are subdivided in meaningless parts and intellectual assembly line work is created. Contrary to that, good systems have the potential to reduce stress (by being forgiving towards errors from the user; e.g. by providing an UNDO command for every operation) and to augment human capabilities. Ergonomics research should indicate wrong developments in an early state; this requires that evaluation of a system should start at the beginning of the development process and should not be restricted to acceptance testing. The use of computer systems as tools is a complex skill which is acquired over long periods of time and it is very costly to untrain people after they have become familiar with one way of doing something.

Ergonomics research faces one critical problem: technology has progressed so fast that some of the questions which were too closely tied to a certain technology became obsolete before the studies were completed. A detailed understanding of the optimal design of a keypunch is of little value today, because keypunches are dying out anyway. The question whether a terminal or a printer should support small *and* capital letters is not discussed any more, because all products which do not are driven out of the market.

When is ergonomics research most important? Applying criteria from ergonomics and psychology to the *acceptance testing* of a given system is definitely easier than applying evaluation criteria to the *design* of systems. In acceptance testing, the system is given; all of its parts and properties are specified. In design, the system is still largely hypothetical; it is a class of systems. But there is much more leverage in evaluation of the system design than in acceptance testing of the finished product. The long ranging goal is that the designer of a complex computer system has a design handbook (similar to the handbooks which are available to the civil engineer today) which gives him access to the important design criteria. Unfortunately this is not the case. We are lacking cognitive theories which are prescriptive enough to be used as design criteria and of planning and a detailed understanding of the specific task structure. In addition, productivity gains inherent in new technology cannot be realized simply by carrying out existing operations at a faster pace; a careful analysis of work procedures is crucial for a substantial improvement.

Why not just ask the user? There is no doubt that the user community should play an active part in the design of a system (shared system design among users and designers) -- but unfortunately most potential user communities can not articulate in enough detail the characteristics of the systems which they would like to have. Therefore it is important that there are research places where users can explore and criticize new system designs in a prototypical development stage.

7. Problems and Challenges for Research in Intelligent Systems

What can we do without complete specifications? System design belongs to the "Sciences of the Artificial" [Simon 81]. Contrary to natural scientists who are given a universe and seek to discover the laws, the system designer makes "laws" in the form of programs and the computer brings a new universe to life. Many interesting problem areas (especially the domain of intelligent systems) contain mostly ill-structured problems. The main difficulty in these domains is not to have a "correct" implementation with respect to given specifications, but to develop specifications which lead to effective solutions which correspond to real needs. Correctness of the specifications is in general no meaningful question because it would require a precise specification of intent, and such a specification is seldom available.

We have to accept the empirical truth that for many tasks system requirements cannot be stated fully in advance - in many cases not even in principle because neither the user nor anyone else knows them in advance.

The development process itself changes the user's and designer's perceptions of what is possible, increases their insights into the application environment, and indeed often changes the environment itself.

The following (not mutually exclusive) possibilities exist to cope with this situation:

1. Development of *experimental programming systems which support the coevolution of specifications and implementations.* Prototypical implementations allow us to replace anticipation (i.e. how will the system behave) with analysis (i.e. how does it actually behave), which is in most cases much easier. Most of the major computing systems (operating systems, editors, expert systems, software development systems) have been developed with extensive feedback (based on their actual use) which continually contributed to improvements as a response to discrepancies between a system's actual and desired state.
2. *Heavy user involvement and participation in all phases of the development process.* The user should be able to play with the preliminary system and to discuss the design rational behind it. An existing prototype makes this cooperation between designer and user much more productive, because the users are not restricted to reviewing written specifications to see whether or not the system will satisfy their needs with respect to functionality and ease of use.
3. *Let the end-users develop the systems.* This would eliminate the communication gap altogether. They are the persons who know most about the specific problems to be solved and by giving them the opportunity to change the system there is no longer a necessity to anticipate all possible future interactions between user and system. *User tailorability* is a first step towards *convivial systems* [Fischer 83] which give the users the capability to carry out a constrained design process within the boundaries of the knowledge area modeled. A strong test for the intelligence of a system 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 foresee.
4. Accept changing requirements as a fact of life and do not condemn them as a product of sloppy thinking; we need *methodologies and tools to make change a coordinated, computer-supported process.*

Understanding the limitations of systems. A critical problem can be that the users do not have a clear understanding of the limitations of a system. This is an important problem for natural language interfaces where users are often seduced to believe that they can talk to these systems like to their colleagues. Users will be especially disappointed if the projected illusion is that of intelligence but the reality falls far short (the ELIZA system may serve as a good example). Despite progress in the area of intelligent systems, it is still the user who is more intelligent and can be directed into a particular context. Giving the user the appropriate cues is one of the essences of human-computer communication. Windows, menus, spreadsheets, documentation, help, explanations, critics and visualization provide a context that allows the user's intelligence to keep choosing the appropriate next step.

Towards Human-ProblemDomain Communication. Despite the fact that we have used the term Human-Computer Communication throughout this paper, we believe that the goal to achieve is Human-ProblemDomain Communication, where the users can deal with descriptions within their world of expertise. Knowledge-based systems are one important step towards this goal and allow users to deal with content instead of form or low-level mechanisms.

Human-oriented computer systems. There is no difficulty in getting people to use computers

provided the computer and its application are genuinely useful. The goal of intelligent systems has to be to create human-oriented computer systems where the users have control and can achieve *their* goals in *their* way. These systems should eliminate the necessity that the human has to become a computer-oriented person (see Figure 7-1).

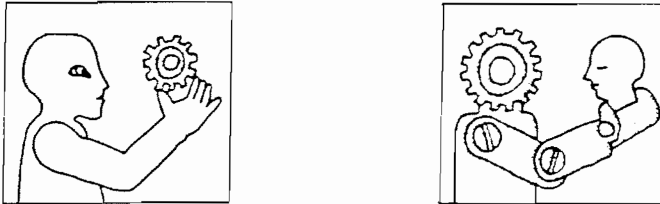

Figure 7-1: Human-centered and computer-centered system
Design of Peter Hajnozcky, Zuerich

References

[Barstow, Shrobe, Sandewall 84]
D. Barstow, H. Shrobe, E. Sandewall.
Interactive Programming Environments.
McGrawHill, New York, 1984.

[Boecker, Fischer, Nieper 85]
H.-D. Boecker, G. Fischer, H. Nieper.
The Enhancement of Understanding through Visual Representations.
Technical Report, University of Colorado, Boulder, 1985.

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

[Boecker, Nieper 85]
H.-D. Boecker, H. Nieper.
Making the Invisible Visible: Tools for Exploratory Programming.
In *Proceedings of the First Pan Pacific Computer Conference.* The Australian Computer Society, Melbourne, Australia, September, 1985.

[Fischer 81[G. Fischer.
Computational Models of Skill Acquisition Processes.
In R. Lewis, D. Tagg (editor), *Computers in Education*, pages 477-481. 3rd World Conference on Computers and Education, Lausanne, Switzerland, July, 1981.

[Fischer 83] G. Fischer.
Symbiotic, Knowledge-Based Computer Support Systems.
Automatica 19(6):627-637, November, 1983.

[Fischer 84] G. Fischer.
Formen und Funktionen von Modellen in der Mensch-Computer Kommunikation.
In M.J. Tauber (editor), *Psychologie der Computernutzung.* Oldenbourg Verlag, Wien - Muenchen, 1984.
Schriftenreihe der Oesterreichischen Computergesellschaft.

[Fischer, Lemke, Schwab 84]
G. Fischer, A. Lemke, T. Schwab.
Active Help Systems.
In T.Green, M.Tauber, G.van der Veer (editor), *Proceedings of Second European Conference on Cognitive Ergonomics - Mind and Computers, Gmunden, Austria.* Springer Verlag, Heidelberg - Berlin - New York, September, 1984.

[Fischer, Lemke, Schwab 85]
G. Fischer, A. Lemke, T. Schwab.
Knowledge-Based Help Systems.
In *Human Factors in Computing Systems.* CHI-85 Conference Proceedings, 1985.

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

[Forgy 81] C.L. Forgy.
OPS5 User's Manual.
Technical Report CS-81-135, CMU, 1981.

[Goldberg 84] A. Goldberg.
SMALLTALK-80, The Interactive Programming Environment.
Addison-Wesley, Reading, Ma., 1984.

[Laubsch, Rathke 83[
J. Laubsch, C. Rathke.
OBJTALK: Eine Erweiterung von LISP zum objektorientierten Programmieren.
In H.Stoyan, H.Wedekind (editor), *Objektorientierte Software- und Hardwarearchitekturen,* pages 60-75. Teubner Verlag, Stuttgart, 1983.

[Lemke 85] A. Lemke.
ObjTalk84 Reference Manual.
Technical Report CU-CS-291-85, University of Colorado, Boulder, 1985.

[Lemke, Schwab 83[
A. Lemke, T. Schwab.
DOXY: Computergestuetzte Dokumentationssysteme.
1983.
Studienarbeit Nr. 338, Institut fuer Informatik, Universitaet Stuttgart.

[Norman 82[D. Norman.
Some Observations on Mental Models.
In D. Gentner, A. Stevens (eds.) (editor), *Mental Models.* Erlbaum, Hillsdale, N.J., 1982.

[Rathke, Lemke 85]
Ch. Rathke, A. C. Lemke.
ObjTalk Primer.
Technical Report CU-CS-290-85, University of Colorado, Boulder, February, 1985.
Translated by V. Patten and C. Morel.

[Sheil 83] B.A. Sheil.
Power Tools for Programmers.
Datamation , February, 1983.

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

[Wilensky et al. 84]
R. Wilensky, Y. Arens, D. Chin.
Talking to UNIX in English: An Overview of UC.
Communications of the ACM 27(6):574-593, June, 1984.