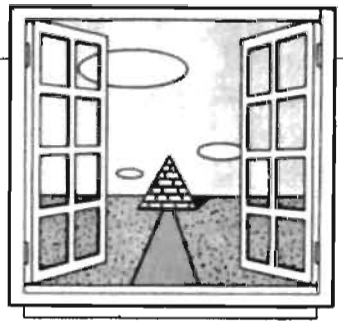

IEEE Software

Human-Computer Interaction Software: Lessons Learned, Challenges Ahead

Gerhard Fischer, *University of Colorado at Boulder*

HCI software should augment human intelligence. Despite the progress that has been made over the last few years, the big challenge is to create truly cooperative problem-solving systems.

Writing *good* software for human-computer interaction is a major challenge. This very new field is based on three developments: Powerful workstations with bitmapped screens and pointing devices provide the new technological base. Innovative applications have drawn attention to the computer's interactive, rather than its computational, capabilities. The complexity of today's software has made better communication techniques a necessity, not a luxury.

Good HCI is important for the products that the software engineer designs and implements for users, but it is equally important for software engineers themselves. They have to design, understand, and maintain complex artifacts that are opaque and very difficult to deal with. Software engineers and programmers have a stake in improving today's environments.

Software design as a communication process

Having spent the last decade building knowledge-based systems, improving human-computer interaction, and exploring the nature of design processes, my colleagues and I are convinced that current life-cycle models of software engineering are inadequate for most of today's computing problems.[1] They are inadequate because they rest on the assumption that problem requirements can be stated precisely at the beginning of a project and that complete specifications and an implementation can be derived from them through formal manipulations.

Software engineering is a design activity. Most design problems are ill-structured[2] and must be solved by exploration and error elimination, considered the foundation of

0704-7459/89/0001/0044/$01.00 © 1989 IEEE

all scientific activity by the philosopher Karl Popper.[3] This is especially true for HCI software: There are no complete HCI theories. The problem is ill-structured, and so many of the methods and tools developed for traditional software-engineering problems are of little use for HCI software.

Ill-structured problems are also characterized by design instabilities and the need for frequent redesign. The main goal of HCI software is not to develop a correct implementation of given specifications but to develop an effective solution that corresponds to real needs. Specification correctness in this context is not a meaningful concept because it implies a precise specification of intent, which is seldom available.

Unless we match our paradigms and methods to ill-structured problems, the implementation disasters of the 1960s will be succeeded by the design disasters of the 1980s. We need effective exploratory programming environments and rapid-prototyping tools that support iterative, evolutionary design.

The best paradigm for creating HCI software is a communication model[1] and a rapid-prototyping approach that supports the coevolution of specification and implementation.[4] Communication between customers, designers, and implementers and between humans and the knowledge base that describes the emerging product is crucial.

Our work at the University of Colorado at Boulder centers on knowledge-based systems that enhance and support communication. When we write HCI software, we define what the computer will do and what humans will and can do. We also make assumptions about what they want to do. It is this human element that distinguishes HCI software:

- Humans are individuals with different talents, goals, knowledge, and preferences.
- Humans are moving targets, not static objects. They start as novices and may remain in this class, but they may also become casual or expert users.

HCI software design must start with the human as a fix point. Humans do not have three hands, and our designs must take this into account — users cannot keep two hands on a keyboard and one on a mouse. HCI software should acknowledge human weaknesses (limited short-term memory and execution errors) and exploit human strengths (a powerful information-processing and visual system).

Knowledge-based HCI

Our research has three dimensions: theory development, engineering construction, and empirical studies. The goal of theory development is to understand how human cognition and design capabilities result from an interplay between mental processes and external computational and memory aids. The purpose of engineering construction is to design and develop the desired artifacts. The empirical research examines how people perform tasks using the artifacts in a natural setting.

The primary application of our research has been the creation of knowledge-based HCI software to run on high-functionality systems in support of cooperative problem solving.

Architecture. Effective HCI is more than creating attractive displays: You must give the computer a considerable body of knowledge about the world, users, and communication processes.

However, the use of knowledge-based systems today is limited severely by the communication bottleneck in the narrow channel between the user and the system. A good user interface is vital to a knowledge-based system, but it has little use if its sophisticated graphical facilities lack rich, supporting information structures. Knowledge-based facilities have elaborate, interactive graphical interfaces supported by specialized viewers and information-management systems.

To support these systems, we have developed a model, illustrated in Figure 1. Our system architecture reflects two major departures from traditional approaches: The use of windows, menus, and pointing devices widens the explicit communication channel, and shared knowledge establishes an implicit communication channel.

Explicit channel. Large bitmapped screens with windows, menus, pointing devices, and speech output and input have widened the explicit channel between the user and computer dramatically. These technologies are necessary but by no means guarantee good HCI.

Exploiting these technologies to benefit the user requires a deeper understanding of psychological principles — for example, the difference between recall and recognition memory, by which menu sys-

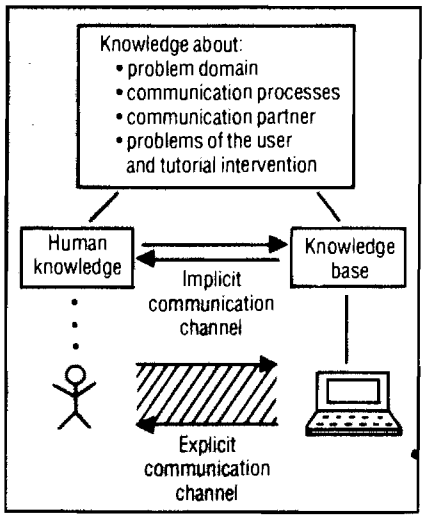

Figure 1. Architecture for knowledge-based human-computer interaction.

tems should be evaluated. We must also take into account the nature of different tasks and the user's abilities.

The design space of the explicit channel is huge, largely unstructured, poorly understood, and inadequately supported by tools. It should therefore come as no surprise that even with improved technical capabilities many HCI systems are still modeled on teletype interaction.

Implicit channel. When communication is based on shared knowledge structures, it is not necessary to exchange all information between the user and system explicitly. Figure 1 lists four knowledge domains necessary for implicit communication:

1. Problem domain. Intelligent behavior builds on large amounts of knowledge about specific domains. This knowledge constrains the number of possible actions and describes reasonable goals and operations. We can infer the user's goals and intentions if we understand the correspondence between the system's primitive operations and the concepts of the task domain. If the computer has a model of the problem domain's abstractions, communication between the human and the problem domain is feasible.

2. Communication processes. The information structures that control communication should be made explicit so the user can manipulate them. Safe, exploratory environments should be supported (for example, with undos, redos, and history lists). User and computer should communicate by writing on a shared display or by referring to something already on the display. This interreferential, I/O model (for example, as supported by the Symbolics presentation system) supports dialogue rather than issuing isolated messages.

3. Communication partner. *The* user of a system does not exist; there are many kinds of users, and an individual's requirements change with experience. Adaptive systems change their behavior according to the needs of different users. Unless the system has a model of the user, it is impossible for it to relate descriptions of new systems to the existing knowledge of individual users.

4. Common problems and instructional strategies. If the system is to be a good coach or teacher and not just an expert, it must incorporate instructional strategies based on pedagogical theories and exploit the knowledge contained in its model of the user. For the same reason, an intelligent support system should know when to interrupt a user.

Object-oriented architectures. Our systems use an object-oriented architecture, which has been shown to be well-suited to interface construction. In this model, the user communicates with the system, which is represented on the screen as a world composed of active objects. Each screen object has its visual representation (which defines its appearance and its relation to other screen objects) and a functional role (which governs its behavior).

The principles of object-oriented design — inheritance, flexibility, extensibility, and modularity — support new programming methodologies that encourage designing for reusability and redesign, such as differential programming and programming by specialization and analogy. Object-oriented architectures provide substantial support for interface design.

The principle of human problem-domain communication abstracts the domain's operations and objects and builds them into the computing environment.

Domain communication. Most users are experts in some problem domain, whether it be physics or music. The computer is a generalist; its generality means it can support all knowledge workers. But domain experts are not interested in learning the computer's languages; they simply want to use it to solve problems and accomplish tasks.

To shape the computer into a truly useful medium, we have to make it invisible and let users work directly on their problems and their tasks. To do this we must "teach" the computer the experts' languages by endowing it with the abstractions of different application domains.

The principle of *human problem-domain communication*[5] abstracts the important operations and objects of a domain and builds them into the computing environment. This lets the user operate with personally meaningful abstractions. It is important not to lose the semantics of the domain by reducing it to oversimplified data structures.

WLisp, our user-interface toolkit,[6] and two systems called Framer and Crack support human problem-domain communication in the areas of interface and kitchen design. People can use these systems to do programming by constructing artifacts in the domain instead of writing statements in a programming language. Our experiments have shown that people who use these environments had a sense of accomplishment because they created their own impressive version of something that works but was not difficult to build.

Framer and Crack, described in the box on pp. 48-49 are design environments (with embedded construction kits): They provide a set of building blocks that model a problem domain. The building blocks define a design space (the set of all designs that can be created by combining these blocks) and a design vocabulary. The advantage of design environments is that they eliminate several prerequisite skills, thus letting users spend much more time working in their area of interest.

The disadvantage is that these design environments are effective in only one area. But this limitation affects all knowledge-based systems, and human expertise is also restricted to specific domains. The challenge for computer systems is to create these design spaces for many domains and to organize this huge set so users can find the abstractions they need.

Reuse and redesign. Software environments must support design methodologies whose main activity is not only the generation of new programs but also the maintenance, integration, modification, and explanation of existing ones.

A construction kit with many general building blocks supports reuse and rede-

sign by providing stable abstractions. The user-interface tools of Smalltalk, the Lisp machines, and WLisp have undergone a long evolutionary development. They are based on abstractions about the domain of interface design and constitute a partial theory of one class of user interfaces. The evolutionary development of such a theory, driven by tests of the abstractions' validity in several applications, is a prerequisite for the development of a system that supports reuse and redesign.

However, standard programming languages offer only a few primitives for HCI (read, write, and format). These primitives are conceptually based on a linear stream, not a two-dimensional screen. So it is a huge effort to build the HCI part of an application because the designer has to build it from low-level components.

On the other hand, functionally rich environments offer hundreds and thousands of abstractions,[6] substrates for HCI (different classes of windows, support for interreferential I/O), and screen-layout tools. Such systems reduce the size of the application system substantially. The major cost of such systems is that the designer must learn and understand the abstractions, but this cost is incurred only once per designer.

Intelligent support systems. High-functionality systems are not without problems, however. Our informal empirical studies have shown that:

- Users do not know that tools exist. It is very difficult for a novice or casual user to build a mental model of the capabilities of a high-functionality system. Without such a model, users are unaware of tools that might be relevant to their tasks. A passive help system is no help in this, because to ask a question the user must know enough to know what is not known! Active help systems, critics, and browsing tools[6] let users explore a system and point out useful information.
- Users do not know how to access tools. Knowing that something exists does not guarantee that you know how to find it.
- Users do not know when to use tools. While each system feature may have a sensible design rationale from the system engineer's viewpoint, this rationale is frequently beyond the user's grasp. Users do not understand which commands are basic and which are advanced, nor do they understand that the options represent different, generalized interaction styles.
- Users do not understand the results that tools produce. Visualization tools and explanation components address this problem.
- Users cannot combine, adapt, and modify tools according to their needs. Even after all the other problems are overcome, in many cases the tool does not do exactly what the user wants. The user needs system support to carry out constrained design processes at the user's operational level.

We have constructed design environments[5,6] to support the modification and construction of new systems from sets of predefined components. In contrast to simple construction kits, our design environments incorporate knowledge about which components fit together and how they do so, and they supply critics that recognize errors and inefficient or useless structures. Our environments can deal with multiple representations of the design.

The success of new computer systems is judged less on processing speed and memory size and more on the quality of their communication capabilites.

Lessons learned

The success of new computer systems (the Macintosh may be the best example) is judged less and less on processing speed and memory size and more and more on the quality of communication capabilities. We have learned that, to be successful, HCI software must:

- Take advantage of technology. Modern workstations provide new technological possibilities. Yet many current user interfaces are still teletype-oriented. It requires a huge design and implementation effort to reconceptualize all software to take advantage of the new possibilities.
- Use user-interface construction kits. User-interface management systems[7] provide graphical primitives and tools to specify dialogue structures, but they limit information exchange and separate the user interface from the application. This is a reasonable approach for some problems. However, for the kinds of problems we try to solve, such as intelligent support for human problem-domain communication, a strong separation between interface and application is not feasible. In these systems, the user interface must have extensive access to the state and actions of the application system. For problems of this kind, user-interface construction kits are more appropriate.

User-interface construction kits can provide powerful environments for rapid prototyping of a large class of interfaces. They provide many building blocks for constructing high-quality interfaces at a relatively low cost, and object-oriented architectures can supply uniformity, extensibility, and incremental development.

User-interface construction kits come close (within their scope) to our notion of human problem-domain communication. Users familiar with problem domains but inexperienced with computers have few problems using these systems, while computer experts unfamiliar with the problem domains could not exploit their power.

The major shortcoming of large construction kits is that they do not help the designer construct interesting and useful artifacts in the application domain. In Crack, for example, it is not enough to provide design units — kitchen design is more than placing appliances. Design environments and critics are needed to help users construct truly interesting artifacts; they surpass construction kits in that they incorporate useful, general knowledge about design.

- Provide exploratory environments. Users do not know what they want, and designers do not understand what users need or will accept. The only viable strategy for HCI software is incremental, evolutionary development. Initial systems must be built to give users something con-

Framer and Crack

Framer and Crack are design environments. Framer supports the construction of window-based user interfaces and Crack supports cooperative kitchen design. Both systems incorporate and illustrate the research issues outlined in this article.

The basic building blocks are application-dependent abstractions that support human problem-domain communication. Framer and Crack incorporate intelligent support systems and offer different design strategies such as design with basic building blocks and redesign of prototypical examples.

Framer. Figure A shows a screen from Framer, an enhanced version of the Symbolics Frame-Up tool. Framer offers the user a palette of domain-oriented building blocks, and supports direct-manipulation interaction in the work area. This visual interaction style is especially appropriate in a domain in which visual objects are designed from visual parts.

In addition to serving as an application-oriented construction kit, Framer has a small rule base with design knowledge about relevant aspects of window-based user interfaces. The Praise command tells a user the positive aspects of a design; the Suggest Improvements command criticizes the design; the Explain option gives some rationale for the suggested improvement.

Framer includes a catalog, which contains several prototypical designs that can be praised and critiqued. When brought into the work area, the user can modify them and use them as a starting point for redesign. Prototypical solutions that can be changed and refined through redesign are important enrichments for designers and enlarge their design possibilities. Users can store their designs in the catalog.

Without Framer, the user would have to write the program in Figure B to generate the screen layout in Figure A. Framer generates this code automatically. An experienced user can then modify the code.

Crack. Critics are another type of intelligent support system. Critics let users pursue their own goals, intervening only if they discover the solution is, according to their knowledge, inferior.

Empirical observations indicate that users are often unwilling to learn more about a system than they need to solve their current problem. To cope with new problems as they arise, a critic must generate advice that is tailored to the user's needs and the current situation. This removes from users the burden of having to learn new things in neutral settings when they do not know if they will ever use them.

Figure C shows a screen from Crack,[1] a critic system that helps users design kitchens. Similar to Framer, it provides a set of domain-specific building blocks and has knowledge about how to combine them into useful designs. It "looks over the shoulder" of users as they design. If it discovers a shortcoming in the design, it offers criticism, suggestions, and explanations and helps users improve the designs through cooperative problem solving. Crack's objective is to blend the designer and the computer into a problem-solving team to produce better designs than either of them could working alone.

User control. Critics in Crack are state-driven condition-action rules that are triggered when nonsatisfying partial designs are detected. These rules are activated after each state change. State changes are all instance creations of design units and of any design-unit manipulation (like move, rotate, and scale). An unsatisfactory solution is an arrangement of design units that violates one or more of the relations between them.

Crack is not an expert system that dominates the design process by generating new designs from high-level goals or resolving design conflicts automatically. Users control the system's behavior at all times and can modify its knowledge base if they disagree with the criticism.

Crack lets the user control the firing of critics at three levels: All critiquing can be turned on or off, individual critics can be enabled or disabled, and specific relations in a critic can be modified. When critiquing is turned off (the default), Crack acts like a construction kit with no design knowledge. When critiquing is enabled, all critics are active. An individual critic can be disabled if a user does not like its criticism or if its knowledge has been assimilated and is no longer needed. At the lowest level, the different relational tests within a critic can be replaced.

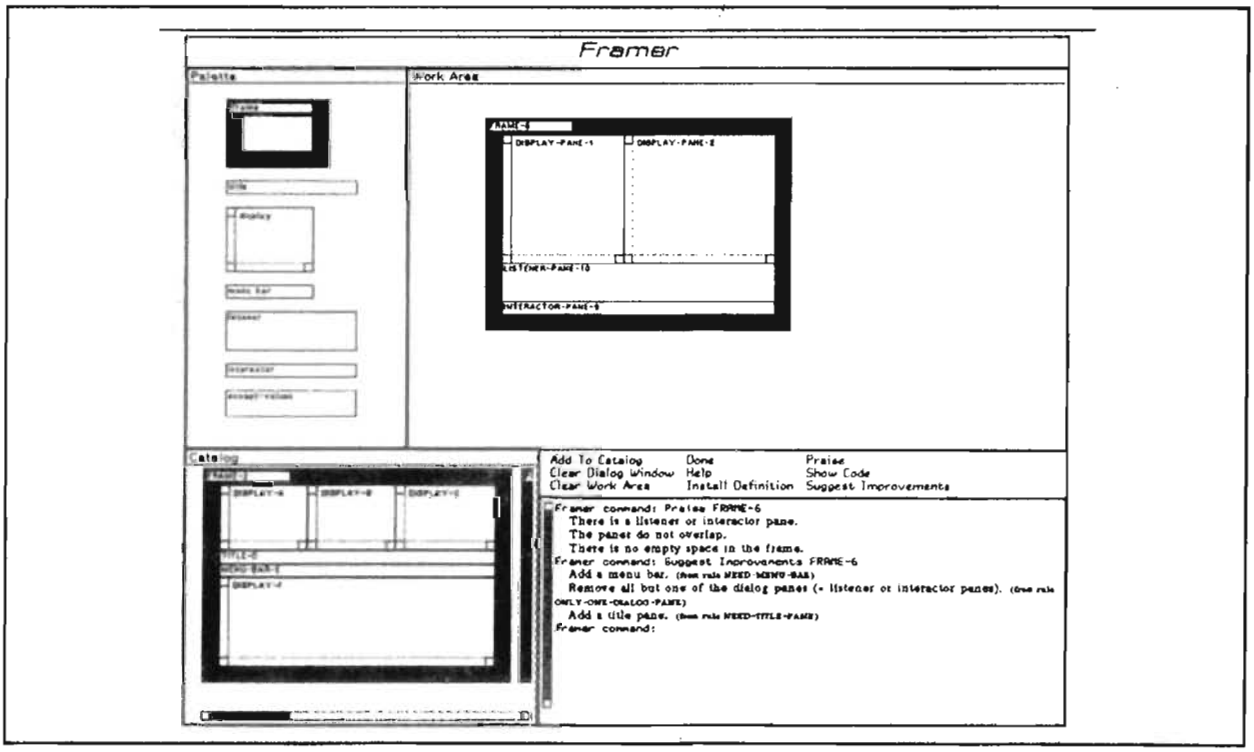

Figure A. A screen from Framer, a design environment for window-based interfaces.

For example, if a user doesn't want to have the sink in front of a window, the in-front-of relation can be replaced with another relation, such as no-relation or close-to.

Appropriate applications. The critiquing approach is best suited for ill-defined problem areas where the goal is to satisfy, rather than optimize, a solution. Kitchen design (as an area of architectural design) is still an ill-defined problem despite the existence of some well-established design principles. Architects do not try to find optimal solutions to design problems but rather make trade-offs within a solution space that is bounded by external constraints.

Critics can be classified as:

• Active or passive. Critics can activate themselves when they detect an unsatisfactory design or they can be passive and wait until the user asks for an evaluation. Active criticism early in the design makes users aware of their unsatisfactory design when the mistake is easier to correct. But users may find it a nuisance to have someone continuously critique them without giving them a chance to develop their own work. A passive critic lets the user request an evaluation when they have completed a partial design. Active critics seem to be suited for guiding novice users; passive critics seem to be more appropriate for intermediate users.

• Reactive or proactive. Crack's critics are reactive: They make comments about what the user has done. A proactive critic plays the role of an adviser by suggesting what the user might do or proposing criteria which the user should consider. For example, a proactive critic in Crack could highlight the area where a new design unit could be located in a partially completed design.

• Positive or negative. Critics can either praise a superior design or complain about an inferior design. Critics in Crack are negative: They complain only about unsatisfactory configurations and do not praise especially useful or interesting configurations. Human critics are both positive and negative.

• Local or global. A critic's granularity determines whether it is oriented toward local aspects of a partial design or global aspects of the total design. A sink critic is a local critic because it is concerned with a low-level design unit: a sink. A work-triangle critic (the work triangle is the center-front distance between the sink, range, and refrigerator) is a more global critic because it is concerned with a larger portion of the design: several appliances. A kitchen critic is a global critic: It is concerned with the look of the entire kitchen.

```
(DW:DEFINE-PROGRAM-FRAMEWORK EXAMPLE-1
   :COMMAND-DEFINER T
   :COMMAND-TABLE
   (:INHERIT-FROM '("colon full command" "standard arguments"
                                         "input editor compatibility")
    :KBD-ACCELERATOR-P 'NIL)
   :STATE-VARIABLES NIL

   :PANES
   ( (PANE-1  :DISPLAY
    (PANE-5  :DISPLAY)
    (PANE-4  :DISPLAY)
    (TITLE  :TITLE  HEIGHT-IN-LINES 1
     :REDISPLAY-AFTER-COMMANDS NIL)
    (|menu bar|  :COMMAND-MENU  MENU-LEVEL
     :TOP-LEVEL)
    (PANE-3  :DISPLAY))

   :CONFIGURATIONS
   '( (DW:MAIN
     (:LAYOUT (DW::MAIN  :COLUMN ROW-1 TITLE
        |menu bar| PANE-3)
              (ROW-1  :ROW PANE-1 PANE-5 PANE-4))
     (:SIZES
      (DW::MAIN  (TITLE 1  :LINES)
                 (|menu bar|  :ASK-WINDOW SELF
                  :SIZE-FOR-PANE |menu bar|)
                 :THEN (ROW-1  :EVEN )(PANE-3  :EVEN))
     (ROW-1 (PANE-1 :EVEN) (PANE-5 :EVEN) (PANE-4 :EVEN))))))
```

Figure B. The Lisp program for the screen layout in Figure A.

Reference

1. G. Fischer and A. Morch, "Crack: A Critiquing Approach to Cooperative Kitchen Design," *Proc. Int'l Conf. Intelligent Tutoring Systems*, ACM, New York, 1988, pp. 176-185.

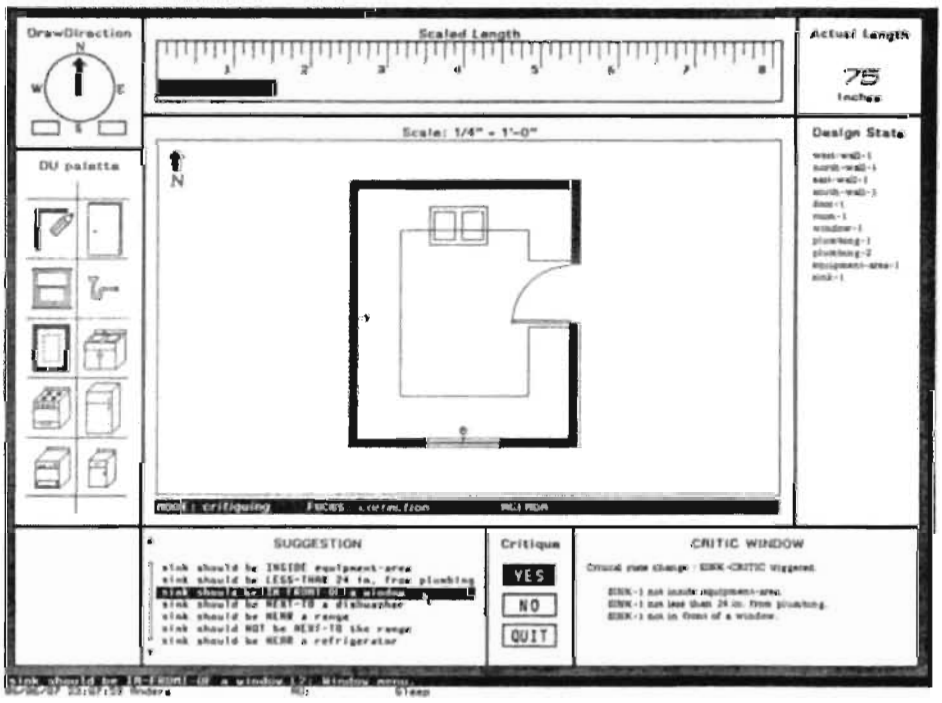

Figure C. A screen from Crack, a critic system for kitchen design.

crete to react to. This strategy must be supported by exploratory programming systems that support the coevolution of specifications and implementations.

Prototypes replace anticipation (how will the system behave?) with analysis (how does it actually behave?), which is much easier to work with. Most major computing systems have been developed with extensive feedback from their actual use. In this way improvements are made in response to discrepancies between a system's actual and desired states.

We know that writers with access to editors and formatting tools are increasingly willing to modify their work. Intelligent support tools in exploratory programming environments will lower the cost of making changes, so designers will also start to experiment — thereby gaining experience and insight leading to better designs.

Exploratory environments also support reuse and redesign. But anyone who thinks that reuse and redesign comes for free is wrong. If reuse and redesign are great ideas and if they are easy to master, why have they had limited success so far in software development? Observation of designers dealing with complex software systems has revealed one reason: These methodologies are not adequately supported. It is too expensive to change a system or explore design alternatives in most software production environments.

• Use prototypes. Static specification languages have little use in HCI software design. First, detailed specifications do not exist. Second, the interaction between a system and its user is highly dynamic and aesthetic, aspects that are difficult, if not impossible, to describe with a static language. Successful HCI systems should let users play with the prototype systems and discuss their design rationale. A prototype makes it much easier and productive for designers and users to cooperate because users do not have to rely on written specifications, which do not indicate an interface's qualities.

• Promote natural communication. Natural communication is more than the ability to communicate in natural language; it is the ability to engage in a *dialogue*. When human novices communicate with human experts, much more goes on than just a request for factual information. Novices may not be able to articulate their questions without the experts' help; the experts' advice may not be understood without further explanation; each may hypothesize that the other misunderstood; experts may provide information they were not explicitly asked for.

Natural communication needs a proper user interface to support it, but it is not restricted to the user interface. The underlying knowledge base must contain the needed knowledge, and it must be structured correctly.

• Help the user make intelligent choices. Communication can be described in terms of the speaker and the listener roles. The speaker presents information, perhaps in the form of a question or as a request for action, which the listener tries to understand. In general, the listener must be the more intelligent agent because he must not only understand a situation as such but also understand how the speaker presents it.

Concern for the human must equal concern for the computer. A theory of human cognitive processes should drive the new communication developments.

In HCI, the user is the more intelligent agent. Therefore, giving the user the appropriate cues is the essence of most HCI designs: The context provided by windows, menus, spreadsheets, property sheets, and form systems lets the user choose the appropriate next step.

Because humans and computers are not alike, designing HCI software is a problem not only of simulating human-to-human communication but of engineering alternatives in the domain of interaction-related properties. Humans do not have a menu on their forehead of the commands they can execute in a certain context.

• Extend the design knowledge base. Why hasn't someone like Donald Knuth written a book with prescriptions for building good HCI software? Because the basis of such a book has yet to be created by cognitive science. We have to extend the existing knowledge base for HCI. Card, Moran, and Newell[8] have provided some methodologies and principles for the design of systems that support routine cognitive skills whose time spans fall within seconds. But these quantitative design methods are not relevant to complex systems, which may take months and years to learn and understand.

An interdisciplinary approach is needed to increase our HCI knowledge base. Concern for the human must equal concern for the computer. A theory of human cognitive processes should drive the development of new communication capabilities. We need design principles for comprehensible systems that are not restricted to the evaluation and assessment of existing HCI systems. If humans are the fix point of future HCI systems, the designs must be based on cognitive principles right from the beginning.

Challenges ahead

User-centered designs, comprehensible systems, intelligent support systems, and powerful tools to augment human intelligence are the main goals of HCI design. The software design community has accepted these goals as relevant, and considerable progress has been made over the last few years. But big challenges lie ahead:

• Focus on innovation. Part of our research effort should be directed toward demonstrating which current systems are good or bad because this gives us insight into the design criteria for future systems. But the main challenge lies in developing new, innovative systems.

Restricting our effort to evolutionary improvement of current systems ignores the history of HCI over the last decades, during which HCI has made revolutionary steps forward. If the cost of developing HCI software is to be reduced, more computer support must be supplied to the development process. Current development occurs not in the computer but in people's heads; it is not documented and therefore cannot be analyzed.

• Perform real evaluations. If the primary approach to HCI is incremental and evolutionary, evaluation of existing sys-

tems is an important activity. The challenge here is not to restrict our evaluations to laboratory experiments but to test real users doing real tasks in real settings. Ultimate criteria are usefulness and usability for real-world purposes and therefore must be tested in the real world. Validation and verification methods from other software domains have limited use in HCI. Formal correctness is crucial, but it is by no means a sufficient measure of the effectiveness and success of an HCI system.

• Accept change. We have to accept the empirical truth that for many systems requirements cannot be stated fully in advance.[2] In many cases, they cannot even be stated in principle because neither designers nor users know them in advance. The development process itself changes the designers' and users' perceptions of what is possible and increases their insights. We have to accept changing requirements as a fact of life and not condemn them as a product of sloppy thinking. We need methodologies and tools that support and coordinate the process of change.

• Tailor software. One crucial shortcomings of computers is that they don't support the notion of accomplishing a task or purpose that the user chooses. To do so would require software that is truly soft, which means that the behavior of a system can be changed without major reprogramming and without specialists. This requirement is critical because predesigned systems are too rigid for problems whose nature and specifications change and evolve.

This goal does not imply transferring the responsibility of good system design to the user. It is probably safe to assume that normal users will never build tools of the quality a professional designer can achieve. But a question for future research is: Should we design the HCI part of a system completely or should we develop metasystems that let users alter the HCI according to their needs? Should systems be adaptive — should they change their behavior based on a model of the user and the task (the idea behind our critic systems) — or should systems be adaptable by the user (as they must be to support reuse and redesign)?

• Provide a window into the knowledge base. It is insufficient for intelligent support systems to just solve a problem or provide information. The user must be able to understand and question their criticism. We assume that users will not ask a program for advice if its expertise cannot be examined. Designers must provide windows into the knowledge base and reasoning processes of these systems at a level that a user can understand. The users should be able to query the computer for suggestions and explanations, and they should be able to modify and augment the critic's knowledge.

An evaluation of Crack demonstrates the relevance of this approach. Crack was used by an architectural designer who considered the cooperative, user-dominated approach to be its most important feature. He felt that this feature set Crack apart from expert-system design tools that often reduce users to spectators. In the current version of Crack, we have deliberately avoided equipping the system with its own design capabilities. Why? Because users increase their knowledge and independence by working with systems that do not do the work for them but that enable them to do it themselves. Too much assistance and too many automatic procedures can reduce users' motivation.

We have to accept changing requirements as a fact of life and not condemn them as a product of sloppy thinking. Tools should support change.

• Improve cooperative problem-solving systems. Current intelligent support systems like Framer and Crack are one-shot affairs. They give advice, but that advice is not a starting point for cooperative problem solving. Human advisory dialogues[9] are judged successful when they allow shared control of the dialogue. Our systems should be designed to manage errors and trouble. No system can be designed to be totally foolproof. The problem in HCI is not simply that difficulties in communicating arise that do not occur in human-to-human communication but that, when the inevitable troubles do arise, the same resources are not available for their detection and repair. Errors should not cause a breakdown of the interaction; they should be regarded as an integral part of the process of accomplishing a task. The goal of a cooperative endeavor is not to find fault or to assess blame but to get the task done.

• Recognize design conflicts. As is true for most real-world design tasks, there is no optimal solution for HCI, only trade-offs. A challenge for the future is to resolve these design conflicts. In our research efforts, we will concentrate on the following trade-offs:

• How do we balance the effort and the large training costs necessary for learning a complex interface with the power of extensive functionality? It is important that we explore techniques that reduce the cognitive costs (such as learning on demand and human-problem communication).

• How do we make computer systems useful and usable at the same time? Useful computers must be complex and have rich functionality because they must model the abstractions of many domains, but it is hard to make complex systems usable. Intelligent support systems promise to partially resolve this design conflict.

• How do we achieve high speed and convenience of use (power) for the experienced user and at the same time achieve ease of learning and use for the novice and the casual user? Adaptive and adaptable systems will more closely accommodate the needs of the individual user.

• How do we separate dialogue management from the application (to enhance reusability and uniformity) and yet retain application-specific semantics? User-interface toolkits, UIMSs, and human problem-domain communication provide different answers to this question.

• How do we ensure the quality of HCI software and minimize the costs of its development? Reuse and redesign will be an absolute necessity for dealing with this design conflict.

• How do we fit new systems comfortably into people's lives and at the same time dramatically change the way they live and work? Incremental learning needs to be

enhanced to expose users to systems that have a low threshold (it is easy to get started) and a high ceiling (the limitations of the systems do not show up immediately).

New formalisms and new tools have helped us augment human intelligence. Computers used in the right way are a unique opportunity to take another big step forward, to create cooperative systems in which humans can achieve more than if they were working alone. HCI software is needed to exploit this potential.

Creating better HCI software will have a major effect on software engineers themselves because their main activity is designing for mostly ill-structured problems. Intelligent computer-support systems and good interfaces are crucial for improving the productivity of the software engineer and for increasing the quality of the software product. The cost of hardware and software in future systems will be small compared with the cognitive costs occurring in the development, comprehension, and use of complex systems. ❖

Acknowledgments

I thank my former colleagues and students on the Inform project at the University of Stuttgart and my current colleagues and students at the University of Colorado, especially Andreas Lemke and Charles Hair, who designed and developed Framer; Anders Morch, who designed and developed Crack; Raymond McCall, who contributed to the evaluation of Crack and who critiqued an earlier version of this article; Janet Grassia and Francesca Iovine, who helped edit the article; and Heinz-Dieter Boecker, Hal Eden, Franz Fabian, Thomas Mastaglio, Helga Nieper-Lemke, Christian Rathke, and Curt Stevens, who all have made major contributions to the research described here.

This research was supported in part by Army Research Institute grant MDA903-86-C0143, US West Advanced Technologies grant 0487.12.0389B, and a Software Research Associates (Japan) grant.

References

1. G. Fischer and M. Schneider, "Knowledge-Based Communication Processes in Software Engineering," *Proc. Seventh Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1984, pp. 358-368.
2. H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., 1981.
3. K.R. Popper, *The Logic of Scientific Discovery*, Harper & Row, New York, 1968.
4. W.R. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Comm. ACM*, July 1982, pp. 438-439.
5. G. Fischer and A.C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction*, No. 3, 1988, pp. 179-222.
6. G. Fischer, "Cognitive View of Reuse and Redesign," *IEEE Software*, July 1987, pp. 60-72.
7. B. Betts et al., "Goals and Objectives for User-Interface Software," *Computer Graphics*, April 1987, pp. 73-78.
8. S.K. Card, T.P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Assoc., Hillsdale, N.J., 1983.
9. J.M. Carrol and J. McKendree, "Interface-Design Issues for Advice-Giving Expert Systems," *Comm. ACM*, Jan. 1987, pp. 14-31.

Gerhard Fischer is professor of computer science and a member of the Institute of Cognitive Science at the University of Colorado at Boulder. Fischer directs the university's Knowledge-Based Systems and Human-Computer Communication Research Group.

His research interests include artificial intelligence, human-computer communication, cognitive science, and software engineering; he is especially interested in bringing these research disciplines together to build cooperative problem-solving systems.

Fischer received a PhD in computer science from the University of Hamburg.

Address questions to Fischer at Computer Science Dept., University of Colorado, Campus Box 430, Boulder, CO 80309; CSnet gerhard@boulder.colorado.edu