Cooperative Software

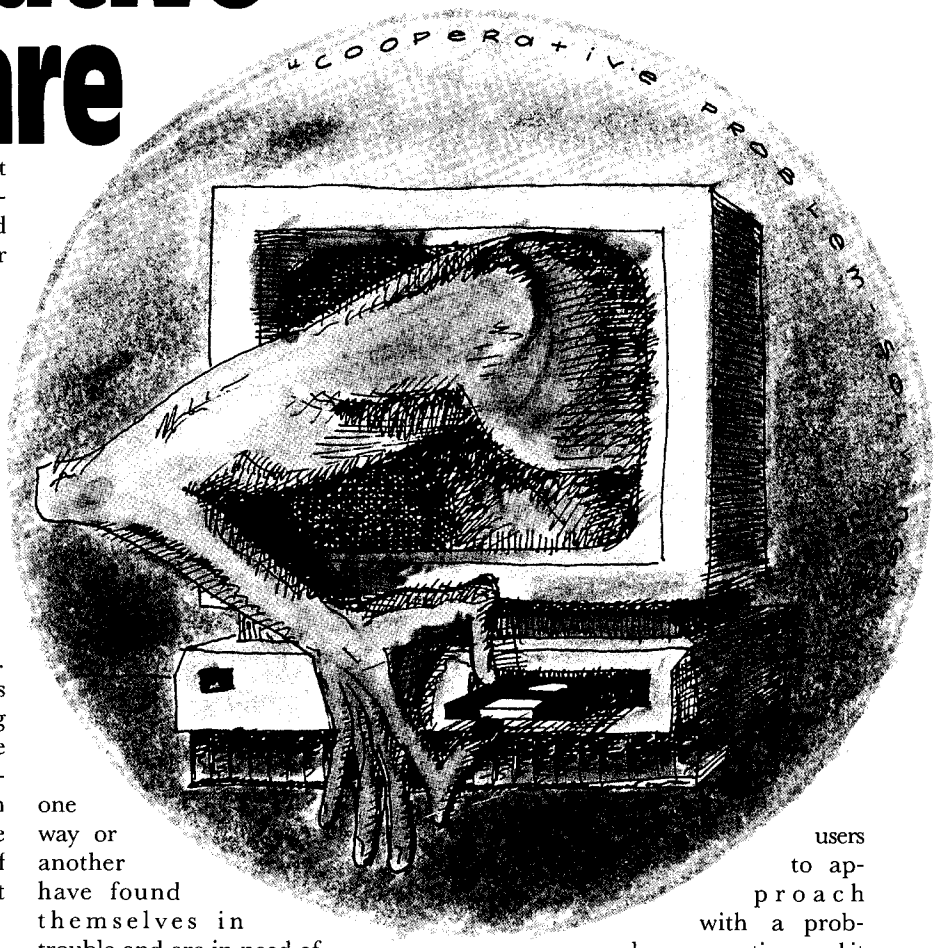

ILLUSTRATION: DENNIS AHLGRIM

There was a problem with my last column ("Hat Racks for Understanding," Oct. 1992, p. 21). It claimed good software helps people in their quest for understanding. Although it mentioned a few strategies for improving a tool's ability to facilitate understanding, we were still left with an elusive and gauzy goal. How do you know when you have understood an object of study? If you looked a little harder or tried another angle, you might discover some hidden quality—a quality that reveals the object's true nature. Then again, maybe not.

Anything so hard to measure presents a problem for technology. While "facilitating understanding" is a worthy goal, it is a poor guiding principle of design. As designers, we can come up with ways to help people examine their data and learn about it, but since we cannot measure understanding it is difficult to tell if we have done our job well. We do not really understand understanding.

Treating Stupidity

Neil Postman, in his tremendous book, *Conscientious Objections*, points out that other fields have the same problem. In his essay, "The Educationist as Painkiller," he observes that physicians do not completely understand health, so they give their attention to relieving sickness—curing disease, halting its spread, and treating symptoms. The same can be said of lawyers, Postman points out, who are not consulted to "improve the quality of justice or good citizenship." Instead, they trouble themselves about injustice and bad citizenship, "of which they know more than anyone else, and which, it turns out, are much more profitable fields of enterprise. Doctors and lawyers, in other words, are painkillers. They are sought out by people who in one way or another have found themselves in trouble and are in need of remedies."

Postman goes on to suggest that educationists abandon their "vague, seemingly arrogant, and ultimately futile attempts to make children intelligent, and concentrate...on helping them avoid being stupid. ...The educationist should become an expert in stupidity and be able to prescribe specific procedures for avoiding it."

Perhaps it will be fruitful for software designers to think about curing ignorance and stupidity. It will be easier to invent treatments and to measure our success.

Some software does try to act as a "painkiller," in that it expects its users to approach with a problem or question, and it responds with whatever its designers thought would be a treatment for that need. The typical database front end is an example.

On the other hand, general purpose software, like spreadsheets, are very powerful tools for understanding. That is, if you already understand the subject area well and know how to use the tool. Otherwise there is a large gap between "where you are" and "where you want to be." To find an answer to your question you must accurately formulate the problem in the tool's terms, select from its large inventory of analysis tools, and know how to employ those tools correctly. No wonder so many people sit staring blankly at their screen, or curse their manuals and help systems, or build inadequate solutions that tell them lies.

Marc Rettig

There are many angles to this problem, one of the great problems of these early days of computing. The best "cures for stupidity" on the market are not general purpose tools, but task-specific applications with a large amount of domain knowledge built into them. Chipsoft's MacInTax is a good example. It knows what all the IRS forms look like, it contains the instructions for every form and entry blank, it knows what it means for a form to be complete, and what kinds of things might trigger an audit. It can lead tenderfoot tax-filers through the entire process (it knows what forms to fill out, and in what order), and can coach or remind experienced labor slaves. The world would be a better place if more software treated people with as much consideration as MacInTax.

This is where the "software as painkiller" analogy leads. Why not make more tools like this? Why not build more domain-specific tools that contain enough knowledge to support people in the problem-solving process—to become a work partner?

One group who is making important contributions to this area has Gerhard Fischer of the University of Colorado as a prolific spokesperson. He and his colleagues have given us a body of literature on "domain-specific cooperative problem-solving systems" well-stuffed with good ideas, fruitful points of view, and working examples. Their systems could be characterized as attempts to create cures for stupidity, in that they try to bolster human weaknesses and support human strengths. (For the sake of readability, I will refer to Fischer when discussing this work, although he usually coauthors with colleagues. I do not mean to discount the contributions of others who are listed in the bibliography at the end of the column.)

Use People's Strength, Support Their Weakness

The term "cooperative problem-solving" means the software and the person using it are partners in the task-at-hand, bringing complementary strengths and weaknesses to the job. (Recall Brenda Laurel's suggestion, in *Computers as Theater*, that designers think of a system as a collection of agents, some human, others software. Laurel and Fischer don't cite each other, but I expect they might like one another's work.)

Fischer observes that the strengths and weaknesses of computers and humans complement each other nicely. "Cooperative problem-solving approaches exploit the *asymmetry* of the communication process. Humans use common sense, define the common goal, decompose problems into subproblems, and so on. Computers provide external memory for the human, insure consistency, hide irrelevant information, and summarize and visualize information. . . . Cooperative problem-solving systems serve as cognitive amplifiers of the human." Strong artificial intelligence isn't necessary for a really intelligent solution. Instead, a team made up of a person's natural intelligence enhanced by good computer software may be cheaper, more effective, and more fulfilling to use.

The screen shot in Figure 1, from a system called Framer, illustrates these ideas. Framer is a tool for designing "program frameworks"—components of user interfaces on Symbolics Lisp machines. The software contains a knowledge base of design rules called "critics." The rules evaluate "hard" aspects of a design, such as its syntactic correctness and completeness, as well as "soft" aspects, such as its conformance to recommended style guidelines. Each critic rule is classified as mandatory or optional, and each is associated with an explanation that explains the motivation behind the rule and suggests ways to achieve the desired effect.

The bottom two panes are where the user carries on the work of building a framework, by dragging items from the palette on the right into the work area on the left. The rest of the screen is where Framer supports the user with various "cures for stupidity." The checklist in the upper left breaks the task of building a program framework into several subtasks. It shows which steps are complete or in progress, and highlights the step currently being performed. The top right pane shows the goal for the current step, and lists the possible actions. The middle pane is where the critics do their stuff. Framer has noticed some problems with the design as it stands, and is offering suggestions.

Fischer is working to improve both the usefulness and usability of what he calls "high-functionality computer systems." He points out that software tools are offering increasing numbers of functions, which present problems for both designers and users. He compares Pascal, with its 29 functions, 19 infix operators, and 300 pages of documentation, with Symbolics Lisp, which has tens of thousands of functions, methods, and special control structures all documented in 4,400 pages. "The more powerful systems become, the more difficult they are to use. Before users will be able to take advantage of the power of high-functionality computer systems, the cognitive costs of mastering them must be reduced." Fischer lists four problems of high-functionality systems:

- Users do not know about the existence of tools
- Users do not know how to access tools
- Users do not know when to use tools
- Users cannot combine, adapt, and modify tools according to their specific needs

Even from this superficial look at Framer, you can see how it addresses these problems. Whatever you think about the aesthetics of the interface, this is a good example of a design that seeks to facilitate understanding, that supports people's weaknesses and enhances their strengths.

Inside a Cooperative Environment

Fischer and his colleagues have experimented with a series of working systems and spent a lot of time watching people in problem-solving situations. They even hung around a hardware store (a high-functionality system) and taped conversations between customers (users) and clerks (domain experts). From this experi-

ence they have evolved a general architecture for domain-oriented, cooperative problem-solving systems, which is worth an overview so you can see how different it is from the applications the rest of us have been building. The architecture consists of five components which contain the domain knowledge and support cooperative problem-solving:

The *construction kit* is the workspace in which people build their design, and the palette of available components. This corresponds to the entirety of what we are used to calling "applications." We are used to delivering a tool and a manual, then waiting for our pat on the back. In Fischer's eyes this is not enough, at least for design-intensive tasks.

The *specification component* lets designers specify some of the high-level requirements of the design, and weight their importance. For example, when using Fischer's Janus to design a kitchen, you could tell its specification component you are only 5'4" tall, you are left-handed, you don't want a dish washer, you have small children (safety is important), and you value energy efficiency.

An issue-based *argumentative hypermedia* system captures knowledge about the domain—especially answers to all aspects of the question, "What makes a good design?" It holds issues, answers, and arguments, with enough description for people to understand the reasoning behind the system's suggestions. All this is linked into a hypertext: terms are defined, cross-references are active, and items can point into the construction, specification, and catalog components.

The *catalog* is a collection of design objects—partial and complete designs, and construction "idioms." They serve as starting points, examples, and jolts to the creativity gland. The *simulation component,* a complement to the argumentation component, lets people try out their designs to see how they will work in various usage scenarios.

These components are integrated by tools that draw from and link together the knowledge, and apply it in context:

The *construction analyzer* detects and critiques the contents of the construction kits' workspace. Critiques are based on domain knowledge of design principles. When a critic fires, it puts a message in the critic pane, and provides entry into the relevant information in the argumentation component.

The *argumentation illustrator* helps people understand the principles in the argumentative hypertext by offering designs from the catalog as concrete examples.

The *catalog explorer* is a sort of browser or front end for the catalog, which finds examples similar to the current construction, and orders them according to their appropriateness for the situation.

Fischer's Powerful Ideas

Fischer writes at length about the

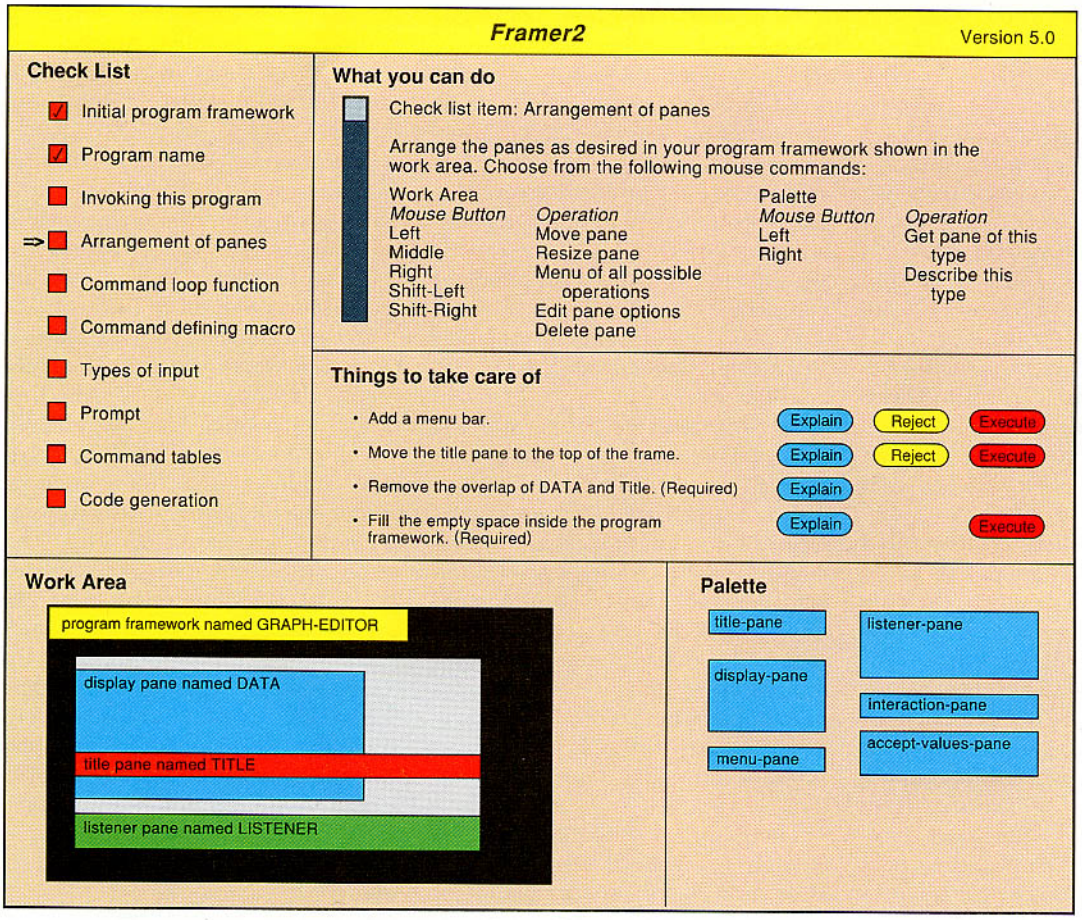

Figure 1. A cooperative tool called Framer, in the process of helping someone construct a window.

Domain-oriented design environments reduce the gap between the knowledge in a designer's head and the language of the tools.

ideas behind these systems, the domain-independent architecture underlying them, and the design principles which they embody. I have room here only to summarize some of the ideas.

Domain Orientation. Domain-oriented design environments reduce the gap between the knowledge in a designer's head and the language of the tools. If the important concepts and constructs of the domain are represented inside the tool, work becomes a matter of "human problem-domain communication," as opposed to human-computer communication. Instead of encoding a design in a symbolic language, people will feel as if they are constructing a design from a set of intelligent building blocks, learning and clarifying as they go.

Critics and "Back Talk." A tool can talk back to the person using it, through "critics." Critics are feedback-generating rules that examine the work in progress and the domain knowledge to offer nonintrusive constructive critiques. Some critics detect problems in the work, some detect inconsistencies between the work and the specifications, and others help designers examine their work from different points of view.

Sometimes it is appropriate for the system to volunteer criticism. Other times it is better to wait until a person asks. Critics support both kinds of dialog, yielding what Fischer calls "mixed-initiative dialog."

Integrate Problem Setting and Problem Solving. Many design problems are poorly understood when the designer sits down to create a solution. Without precise goals, it is difficult or impossible to decide which of the available tools to use. Fischer's systems allow people to propose a partial solution (by choosing from a catalog of parts or starting from scratch), reflect on the "back talk" from the critics, then plan their next move. This is a far more natural process than most systems encourage, and especially valuable for complex systems which evolve through a long, iterative process of prototyping, evaluation, and revision.

Integrate Action and Reflection. Design is an action that proceeds until some sort of breakdown occurs. That is, until the designer hits a stump. Reflection is then used to overcome the breakdown—the designer thinks about the design, the requirements, and the teachings of the trade until a solution is found. Fischer's systems participate in this process. They "talk back," so people can understand, consider, and repair problems the minute they arise. This repair process may trigger new insights, which the designer can add to the tool as a new critic rule or catalog entry.

Integrate Construction and Argumentation. The reflection process—reasoning out a solution to a breakdown in the design process—can be supported by computer-based argumentation tools. That is, a critic not only points out a problem, it provides access to information about the nature of the problem, and the rationale behind it. The tool's domain knowledge should not only include facts and issues, but information about their dependencies and relationships.

As you can see from all this, Fischer advocates a tight integration of the user interface mechanisms with the underlying domain knowledge. This stands in contrast to most modern systems, which separate the interface and knowledge into separate "layers." He gives an analogy to support his claim: "a person who can communicate well but knows very little has severe limitations as a cooperative partner, just as a person who knows a lot but cannot communicate."

There is much more. I hope this is enough for you to see that these ideas are based on a theory of problem-solving, and could be fruitfully applied far outside the uses that the originators have tried.

Performance Support: Learners vs. Students

There is a related, but less formally grounded movement heading in the same direction as Fischer and his colleagues. They call their products "Electronic Performance Support Systems" (EPSS), or just PSS for short. The people building these systems are starting to coalesce into a community of like-minded believers, most of them working in industry (as opposed to universities). The second conference was held last fall in Dallas. This work grew out of efforts to build effective computer-based training systems. Someone realized the best time to provide training is at the very moment the learner needs to apply the knowledge. So the best place for computer-based training is *inside* the tool needed to accomplish the work.

That point bears some elaboration, since it is so different than the way we have been conducting technical training. For other things, like learning to swim, training on demand seems natural to us.

In the June 1981 issue of *National Geographic,* there is a pair of photographs I enjoy very much. The first is an underwater view of a child learning to swim. She is stroking away under water, her instructor swimming alongside observing and providing a sense of security. The second is an old Red Cross photograph of a swimming student engaged in a "land drill." She's equipped for the exercise—hair up on her head and bathing dress draped around her. But the learning environment is not very realistic. She is laying prone across a piano stool in an awkward imitation of a swimmer. She's moving her arms and kicking her feet, but she's not going anywhere.

Instead of encoding a design in a symbolic language, people will feel as if they are constructing a design from a set of intelligent building blocks, learning and clarifying as they go.

I first saw these photos in a book about language learning, on a page entitled "Learner vs. Student." I believe the ideas on that page, paraphrased here, are somehow a part of all great software.

1. The learner is primarily involved with the subject of study and the people who deal with it, in a normal context.

The student is primarily involved with the books and studies of the course, in an isolated study context.
2. The learner revels in the immersion of the real-life experience.

The student is fearful of immersion—"land drills" are more the style.
3. The learner learns, and the very process is a means of communicating interest and care. Learning is doing.

The student studies in hopes of preparing to someday be able to do.
4. The goal of the learner is to accomplish the task.

The goal of the student is usually to "learn" the subject.
5. The attitude of the learner enables one to implement a strategy that will foster a love of doing the task.

The attitude of the student often results in a strategy of comfortable isolation, interrupted with occasional forays into the world of "doers."
6. The learner values the culture and body of knowledge that surrounds the task, seeing it as a bridge to learning.

For the student, the culture and requisite knowledge may be viewed as a barrier, not a bridge.

Performance support systems, taking these ideas very much to heart, try to provide everything a person needs to train him- or herself to do a task on the job, at the moment the training is needed. A PSS goes far beyond a help system or even computer-based training—a large one may have an advisory system, an "information base" of reference material, video and still images, databases, interactive training modules, a help system, preformatted templates and scripts, as well as applications software. The whole thing is designed to guide people through tasks, presenting appropriate resources, and giving just the right amount of training or assistance at just the right time.

This is an ambitious goal, but there are already a dozen or more working performance supports systems in existence, at companies like IBM, Dow, AT&T, Intel, American Express, and Amdahl. Some of these companies are reporting remarkable cuts in training costs, with significant increases in people's performance, and fewer errors. These benefits were gleaned by giving people the opportunity to be "learners" instead of "students"—to immerse themselves in the work, pursue questions on their own, and acquire knowledge on the way to meeting their goals.

Gloria Gery has written a fine overview of the motivation, goals, and philosophy behind performance support systems, and describes many of the working systems in her book, *Electronic Performance Support Systems.* If you are intrigued by anything in this column, I suggest reading her book. She not only deals with technology, but with the large and hard problems of organizational change necessary to develop and use performance support systems.

Nice, but Implementation Is a Bear

Now we have two new goals for our application designs: "facilitate understanding," and "treat stupidity." The areas of conceptual modeling and information design help with the first goal, and the ideas behind cooperative environments and performance support systems help with the second. Integrating these ideas into your work is likely to change the software you design, inside and out. It will also change the process you go through to design and build software. Your customers will have a different role in the process, and they may have to be sold on the benefits of these ideas. Otherwise they will never make the organizational changes they must make to accommodate these systems.

It is not enough to think of computer-based tools as isolated bits of technology we can drop into people's hands, then step back while they reap the benefits. Instead, computers are part of a web of people and tools (good managers have always known this; programmers are just starting to learn). Putting a new item into the web affects the others, and it is silly to design something without considering how it will be used, and by whom.

The technology is only half the solution. The material referenced in the bibliography will give you all kinds of ideas about technology. But to turn these ideas into effective solutions means acquiring good domain knowledge, representing it well, doing good instructional design, information design, human factors, and visual design. It means managing the technology well, shaping it to fit the organization and making sure people know how to use it.

The easy thing would be to pick a few ideas and add them to your existing designs: a to-do pane, critics, mixed-initiative dialogs, some domain knowledge behind the help system, and so on. Adding these things piecemeal may improve your products. But bigger benefits will come from adopting the philosophy behind the ideas and building a whole project around it.

Bibliography

Fischer, G. Domain-oriented design environments. In *Proceedings of the Seventh Knowledge-based Software Engineering Conference.* IEEE Computer Society Press, 1992. pp. 204–213.

Fischer, G., Lemke, A., Mastaglio, T. and Morch, A. The role of critiquing in coop-

erative problem solving. *ACM Trans. Info. Syst. 9,* 2 (Apr. 1991), 123–151.

Fischer, G., Grudin, J., Lemke, A., McCall, R., Oswald, J., Reeves, B. Shipman, F. Supporting indirect collaborative design with integrated knowledge-based design environments. *Human-Computer Interaction,* Vol. 7, Lawrence Erlbaum, Hillsdale, NJ, (1992), pp. 281–314.

Fischer, G. and Reeves, B. Beyond intelligent interfaces: Exploring, analyzing, and creating success models of cooperative problem solving. *J. Appl. Intel. 1,* Kluwer Academic Publishers (1992), 311–332.

Gery, G.J. *Electronic Performance Support Systems.* Weingarten Press, Boston, Mass., 1991. (Weingarten also offers a video demonstration of several EPSS applications.)

Laurel, B. *Computers as Theatre.* Addison-Wesley, Reading, Mass., 1991.

Postman, N. *Conscientious Objections.* New York: Vintage Books, 1988.

Follow-ups and Pointers

Clark, R.C. *Developing Technical Training.* Addison-Wesley, Reading, Mass., 1989. (I didn't mention this book in the column, but it relates directly to the topic and is useful for plenty of things besides training. Clark describes five basic types of content, and tells us how to communicate each type. This is basically the information mapping approach, concisely and directly stated.)

Information Mapping, Inc., 300 Third Ave., Waltham, MA 02154. 607-890-7003. Robert Horn came up with information mapping concepts almost 20 years ago, but the ideas haven't penetrated the software development community very deeply. To hear it from the source, attend an information mapping seminar. All aspects of technical communication are addressed, including computer-based tools.

The Practical Programmer wants to hear your stories. What worked for you, and why? What didn't work, and what were the horrible results? Send your braggardly tales and autopsy reports to:

Marc Rettig
Academic Computing
Summer Institute of Linguistics
7500 West Camp Wisdom Road
Dallas, TX 75236
Email 76703.1037@compuserve.com

***Marc Rettig** is a member of the technical staff at the Summer Institute of Linguistics, and a freelance writer.*