University of Colorado at Boulder

Gerhard Fischer
Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-1502, FAX: (303) 492-2844
e-mail: gerhard@cs.colorado.edu

Putting the Owners of Problems in Charge with Domain-Oriented Design Environments

Gerhard Fischer

Department of Computer Science and Institute of Cognitive Science,
University of Colorado, Boulder, Colorado 80309
email: gerhard@cs.colorado.edu

D. Gilmore, R. Winder, F. Detienne: "User-Centered Requirements for Software Engineering Environments", Springer Verlag, Heidelberg, eds. 1994, pp. 297-306

Abstract: Domain workers should gain considerably more independence from computer specialists. Just as the pen was taken out of the hands of the scribes in the middle ages, the role of the high-tech scribes should be redefined and the owners of problems should be put in charge.

With this goal in mind, we have developed conceptual frameworks, innovative prototypes and an architecture for integrated, domain-oriented, knowledge-based design environments.

Domain-oriented architectures do not only constitute an incremental improvement over current software design practices, but represent a major change to the nature of software development. They reconceptualize our understanding of the proper role of computing as an empowering technology for all of us.

Acknowledgments. The author would like to thank the members of the Human-Computer Communication group at the University of Colorado who contributed to the conceptual framework and the systems discussed in this article. The research was supported by the National Science Foundation under grants No. CDA-8420944, IRI-8722792, and IRI-9015441; by the Army Research Institute under grant No. MDA903-86-C0143, and by grants from the Intelligent Interfaces Group at NYNEX, from Software Research Associates (SRA), Tokyo, and by the Software Designer's Associate (SDA) Consortium, Tokyo.

Putting the Owners of Problems in Charge with Domain-Oriented Design Environments

Gerhard Fischer

Department of Computer Science and Institute of Cognitive Science,
University of Colorado, Boulder, Colorado 80309

Tel: 303-492-1502 Fax: 303-492-2844
Email: gerhard@cs.colorado.edu

Abstract: Domain workers should gain considerably more independence from computer specialists. Just as the pen was taken out of the hands of the scribes in the middle ages, the role of the high-tech scribes should be redefined and the owners of problems should be put in charge.

With this goal in mind, we have developed conceptual frameworks, innovative prototypes and an architecture for integrated, domain-oriented, knowledge-based design environments.

Domain-oriented architectures do not only constitute an incremental improvement over current software design practices, but represent a major change to the nature of software development. They reconceptualize our understanding of the proper role of computing as an empowering technology for all of us.

Keywords: integrating problem setting and problem solving, ill-defined problems, languages of doing, incremental problem formulation, owning problems, domain-oriented design environments, high-tech scribes

1 Introduction

Most current computers systems are approachable only through complex jargon that has nothing to do with the tasks for which people use computers — requiring high-tech scribes (programmers, knowledge engineers) who are able to master this jargon. The role of high-tech scribes should be redefined, eliminating the distinction between programmers and non-programmers as two disjoint classes, and defining programming as the mean for users to make computer do what they want them to do, thereby putting the owners of problems in charge. In this paper, I will (1) identify problems facing user-centered software engineering environments, (2) describe domain-oriented design environments as systems addressing these problems, and (3) assess to what extent we have succeeded or failed in putting the owners of problems in charge.

2 Problems for Future User-Centered Software Engineering Environments

Computing needs to be deprofessionalized. The monopoly of highly trained computing professionals, the high-tech scribes, should be eliminated just as the monopoly of the scribes was eliminated during the reformation in Europe. In order to avoid misunderstandings: This does *not* mean that there is no place for professionals programmers and professional system designers in the future. It means that the profes-

sional computing community should create systems to make computer literacy desirable and achievable. Some of the problems and challenges behind this approach are briefly described in this section.

Convivial tools. Convivial tools and systems, as defined by Illich [Illich 73], allow users "to invest the world with *their* meaning, to enrich the environment with the fruits of *their* vision and to use them for the accomplishment of a purpose *they have chosen*" (emphasis added). Conviviality is a dimension that sets computers apart from other communication and information technologies (e.g., television) that are passive and cannot conform to the users' own tastes and tasks. Passive technologies offer some selective power, but they cannot be extended in ways that the designer of those systems did not directly foresee. Convivial systems encourage users to be actively engaged in generating creative extensions to the artifacts given to them. They have the potential to break down the counterproductive barrier between programming and using programs.

Unfortunately, in most current computer systems the potential for conviviality exists in principle only. Many users perceive computer systems as unfriendly and uncooperative, and their use as too time consuming. They depend on specialists for help, notice that *software is not soft* (i.e., the behavior of a system can not be changed without reprogramming it substantially), and spend more time fighting the computer than solving their problems.

Problems in the Design of Software Systems. The field study by Curtis, Krasner, and Iscoe [Curtis, Krasner, Iscoe 88] unveiled the following problems in creating large software systems: (1) *the thin spread of application domain knowledge*, indicating that the real problem is understanding the problem, not the representation of it as a program, (2) *fluctuating and conflicting requirements*, requiring that the owners of the problems remain part of the design team and that design in use [Henderson, Kyng 91] is indispensable, and (3) *communication bottlenecks and breakdowns* between designers, clients, and users, requiring representational means, such as "languages of doing" [Ehn 88], that can achieve a shared understanding between these groups.

Beyond Programming Languages: From Supply-Side to Demand-Side Computing. Dertouzous (as reported in [Denning 88]) argues that the computer science community should operate less on the supply side (i.e., specifying and creating technology and "throwing the resulting goodies over the fence into the world"). More emphasis should be put on the demand side creating computational environments fitting the needs of professionals of other disciplines outside the computer science community. Modern application needs are not satisfied by traditional programming languages that evolved in response to systems programming needs [Shaw 89; Winograd 79]. Most computer users are interested in results, not in programming per se. Shaw [Shaw 89] claims that "*the major current obstacle to widespread, effective exploitation of computers is the inability of end users to describe and control the computations they need—or even to appreciate the computations they could perform—without the intervention of software experts.*"

Understanding Problems — Beyond Creating Implementations for Given Specifications. Historically, most software engineering developments (e.g., structured programming, verification methods, etc.) were concentrated on "downstream activities" [Belady 85]). Over the last decade, it has become increasingly obvious that the real problems of software design will be "upstream activities" [Sheil 83] (see Figure 1).

While there is growing evidence that system requirements are not so much analytically specified as they are collaboratively evolved through an iterative process of consultation between end-users and software

	Upstream	Downstream
type of problem:	ill-defined problems	well-defined problems
prime objectives:	adequate, understandable, enjoyable	correct
supporting system architecture:	domain-oriented design environments	general-purpose programming environments
interaction paradigm:	human problem-domain communication	human computer communication

Figure 1: Upstream Versus Downstream Activities

developers [CSTB 90], many research efforts do not take this into account. CASE tools are limited, because they devise more elaborate methods of insuring that software meets its specification, hardly ever questioning whether there might be something wrong with the specifications themselves. One may argue that they provide support after the problem has been solved. A consequence of the thin spread of application knowledge [Curtis, Krasner, Iscoe 88] is that specifications often occur when designers do not have sufficient application knowledge to interpret the customer's intentions from the requirement statements — a communication breakdown based on a lack of shared understanding [Resnick 91].

Integrating Problem Setting and Problem Solving. Design methodologists (e.g., [Schoen 83; Rittel 84]) demonstrated with their work the strong interrelationship between problem setting and problem solving. They argue convincingly that (1) one cannot gather information meaningfully unless one has understood the problem, but one cannot understand the problem without information about it [Rittel 84], and (2) professional practice has at least as much to do with defining a problem as with solving a problem [Schoen 83]. New requirements emerge during development, because they can not be identified until portions of the system have been designed or implemented. The conceptual structure underlying complex software systems are too complicated to be specified accurately in advance, and too complex to be build faultlessly [Brooks 87]. Specification and implementation have to co-evolve [Swartout, Balzer 82] requiring that the owners of the problems need to be present in the development. If these observations and findings describe the state of affairs adequately, one has to wonder why waterfall models are still alive despite the overwhelming evidence that they are not suited for most of today's software problems.

In our own work, we have conducted an empirical study in a large hardware store to clarify the dependencies between problem setting and problem solving [Fischer, Reeves 92]. Figure 2 illustrates how the problem setting is changed in an attempt to solve the problem. The customer came to the store to buy a heater. The interaction between the sales agent and the customer led to a reconceptualization of the problem from "generating more heat," to "containing heat" redefining the problem *itself*.

Why Owners of Problems Need to be in Charge. As the previous example shows, "problems are often

CUSTOMER: I want to get a couple of heaters for a downstairs hallway.

SALESPERSON: What are you trying to heat? Is it insulated? How tall are the ceilings? (Remark: They figure out that two of the heaters would work).

CUSTOMER: The reason it gets so cold is that right at the end of the hallway is where the stairs are and the stairs just go up to this great big cathedral ceiling.

SALESPERSON: Well maybe the problem isn't that you're not getting enough heat downstairs, maybe your problem is that you're not keeping the heat downstairs. Do you have a door across the stairs?

CUSTOMER: No.

SALESPERSON: Well that's the problem. You can put a ceiling fan and blow the hot air back down, or cover it up with some kind of door.

Figure 2: Reconceptualizing a Problem: From Generating to Containing Heat

dilemmas to be resolved, rarely problems to be solved" [Lave 88]. Ill-defined problems cannot be delegated (e.g., from clients to professional software designers or professional architects), because the problems are not understood well enough that they can be described in sufficient detail. The owners of the problems need to be part of the problem solving team. Imagine in the above example, the customer would have not gone to the store himself but send someone else with a problem description to buy a heater. This person would have lacked the necessary background knowledge [Winograd, Flores 86] as well as the authority to redefine the problem on the fly.

3 Domain-Oriented Design Environments

In order to put problem owners in charge, future software environments must be able to interact with their users at the level of the task and not only on the level of the medium. Over the last decade, we have designed and evaluated several prototypes addressing this goal. This section will briefly describe the steps leading towards our current version of domain-oriented design environments as well as one example of such an environment.

Towards Integrated, Domain-Oriented Design Environments. The first step towards creating more human-centered computational environments was the development of *general purpose programming* environments exploiting the capabilities of modern workstations. While these environments were powerful and functionality rich, they required users to build their systems from scratch. *Object-oriented design environments* (such as Smalltalk, Clos, C++) represented an effort to create a market place [Stefik 86] for software objects by providing substrates for reuse and redesign at the programming level. Their value is based on the empirical fact [Simon 81] that complex systems develop faster if they can be built on stable subsystems.

But domain-independent object-oriented systems are limited in the support they can provide at the problem level. They consist of low-level abstractions (e.g., statements and data structures in programming languages, primitive geometric objects in computer-aided design, etc.). Abstractions at that level are far removed from the concepts that form the basis of thinking in the application domains in which these artifacts are to operate. The great transformation distance between the design substrate and the application domain is responsible for the high cognitive costs and the great effort necessary to construct artifacts using computers.

Domain-oriented construction kits [Fischer, Lemke 88] intentionally sacrifice generality for more elaborate support of domain semantics. But construction kits do not in themselves lead to the production of interesting artifacts [Fischer, Lemke 88], because they do not help designers perceive the shortcomings of the artifact they are constructing. Artifacts by themselves do often not "talk back" [Schoen 83] sufficiently, except to the most experienced designers. *Critics* [Fischer et al. 91a] operationalize the concept of a situation that "talks back." They use knowledge of design principles to detect and critique partial and suboptimal solutions constructed by the designer.

JANUS: An Example. To illustrate some of the possibilities and limitations of these systems, we will use the JANUS system [Fischer, McCall, Morch 89] as an "object-to-think-with." JANUS supports kitchen designers in the developments of floorplans. JANUS-CONSTRUCTION (see Figure 3) is the construction kit for the system. The palette of the construction kit contains domain-oriented building blocks such as sinks, stoves, and refrigerators. Designers construct by obtaining design units from the palette and placing them into the work area. In addition to design by *composition* (using the palette for constructing an artifact from scratch), JANUS-CONSTRUCTION also supports design by *modification* (by modifying existing designs from the catalog in the work area).

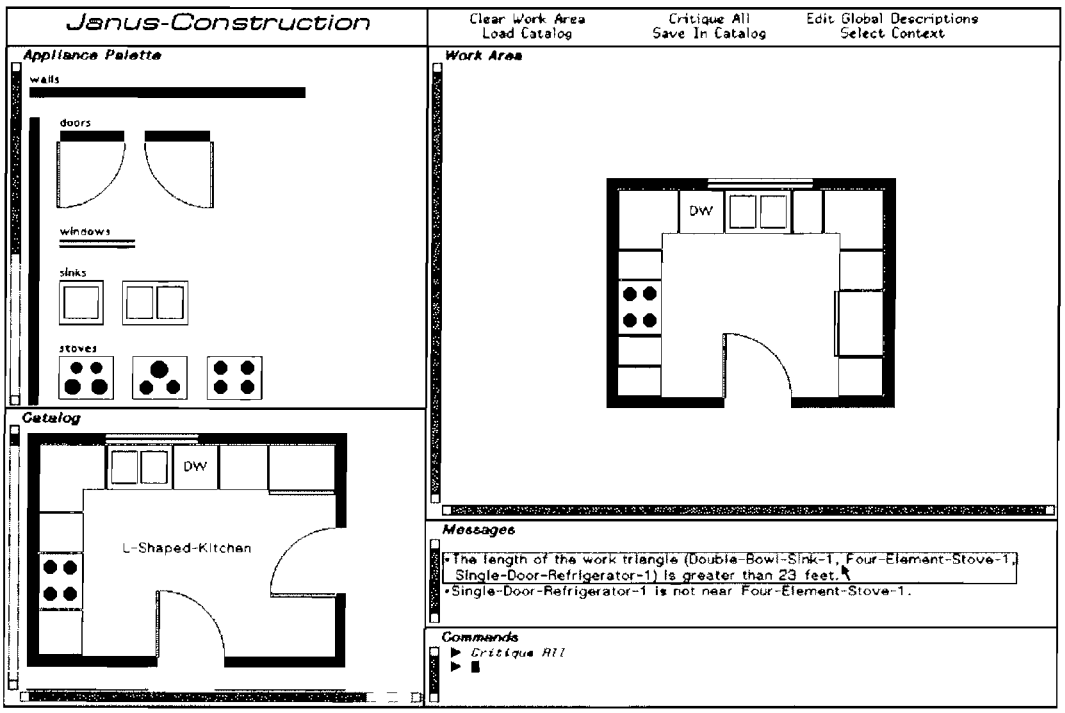

Figure 3: JANUS-CONSTRUCTION: The Work Triangle Critic

JANUS-CONSTRUCTION is the construction part of JANUS. Building blocks (design units) are selected from the *Palette* and moved to desired locations inside the *Work Area.* Designers can reuse and redesign complete floor plans from the *Catalog.* The *Messages* pane displays critic messages automatically after each design change that triggers a critic. Clicking with the mouse on a message activates JANUS-ARGUMENTATION and displays the argumentation related to that message.

The critics in JANUS-CONSTRUCTION identify potential problems in the artifact being designed. Their

knowledge about kitchen design includes design principles based on building codes, safety standards, and functional preferences. When a design principle (such as "the length of the work triangle is greater than 23 feet") is violated, a critic will fire and display a critique in the messages pane of Figure 3. This identifies a possibly problematic situation (a breakdown), and prompts the designer to reflect on it. The designer has broken a rule of functional preference, perhaps out of ignorance or by a temporary oversight.

Our original assumption was that designers would have no difficulty understanding these critic messages. Experiments with JANUS [Fischer, McCall, Morch 89] demonstrated that the short messages the critics present to designers do not reflect the complex reasoning behind the corresponding design issues. To overcome this shortcoming, we initially developed a static explanation component for the critic messages [Lemke, Fischer 90]. The design of this component was based on the assumption that there is a "right" answer to a problem. But the explanation component proved to be unable to account for the deliberative nature of design problems. Therefore, argumentation about issues raised by critics must be supported, and argumentation must be integrated into the context of construction. JANUS-ARGUMENTATION is the argumentation component of JANUS [Fischer et al. 91a]. It is an argumentative hypertext system offering a domain-oriented, generic issue base about how to construct kitchens. With JANUS-ARGUMENTATION, designers explore issues, answers, and arguments by navigating through the issue base. The starting point for the navigation is the argumentative context triggered by a critic message in JANUS-CONSTRUCTION. By combining construction and argumentation, JANUS was developed into an integrated design environment supporting "reflection-in-action" as a fundamental process underlying design activities [Schoen 83].

But even integrated design environments have their shortcomings. Design in real world situations deals with complex, unique, uncertain, conflicted, instable situations of practice. Design knowledge as embedded in design environments will never be complete because design knowledge is tacit (i.e., competent practitioners know more than they can say [Polanyi 66]), and additional knowledge is triggered and activated by situations and breakdowns. These observations require computational mechanisms in support of *end-user modifiability* [Fischer, Girgensohn 90]. The end-user modifiability of JANUS allows users to introduce new design objects (e.g., a microwave), new critiquing rules (e.g., appliances should be against a wall unless one deals with an island kitchen), and (3) kitchen designs which fit the needs of a blind person or a person in a wheelchair.

A Historical Context. Computers in their early days were used to compute. The typical problem at the time was: take a given algorithm and code it in assembly language. The process of programming was totally computer-centered (Figure 4). A large transformation distance existed between the problem description and its solution as a computer program.

High-level programming languages (Fortran, Lisp, Cobol, Algol, etc.) became available in the 1960s. Certain problem domains could be mapped more naturally to programming languages (e.g., algebraic expressions, recursive functions, etc.). While all of the languages remained general purpose programming languages, a certain problem orientation was associated with individual languages (e.g., Lisp for AI, Fortran for scientific computing). The professional computing community started to become specialized: (1) compiler designer (creating programs that mapped from high-level languages to assembly language), and (2) software engineers (creating programs that mapped problems to programming languages).

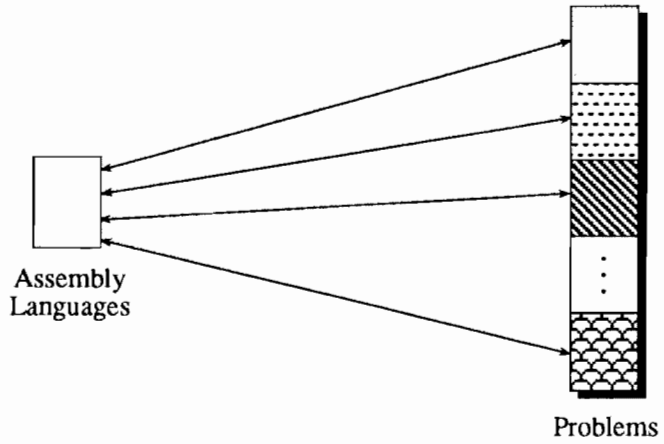

Figure 4: The 1950s: Describing Problems in the Computer's Internal Language

In the 1970s and 1980s new classes of programs appeared: spreadsheets, query languages for databases, and powerful interactive programming environments. Spreadsheets were successful for three major reasons: (1) they relied on a model and packaged computational capability in a form that the user community was familiar with, (2) they added important functionality (propagation of changes) that did not exist in the non-computational media, and (3) they avoided the pitfall of excess generality: Instead of serving all needs obscurely, they serve a few needs well [Shaw 89]. These types of systems can be considered to be construction kits with limited domain-orientation. In our own work, we demonstrated [Fischer, Rathke 88] that the spreadsheet model can be further enhanced by adding additional domain knowledge to it.

By extending construction kits with critiquing and argumentation components, *design environments* (Figure 5) have the potential to put domain experts (the problem owners) in charge. Computational environments based on design environment architectures lead to a further specialization among computer users: knowledge engineers in collaboration with domain workers create design environments (at least the seeds for them [Fischer et al. 91b]) and domain workers use and evolve the seeded environments.

4 Assessment, Evaluation and Implications

Is There an Owner or are There Owners of Problems? So far we used the concept of "ownership" in a way suggesting that there is "an owner" of a problem ignoring that realistic design problems are multi-person activities where ownership is distributed. The reason for this is that neither clients, system designers, nor users have a very good idea of what they want a system to do at the outset of a project. Rittel [Rittel 84] speaks of "a symmetry of ignorance" arguing that knowledge for design problems is distributed among many people, and that there is nobody among those carriers of knowledge who has a guarantee that her/his knowledge is superior to any other person's knowledge with regard to the problem at hand. The notion of "owning" is related to the amount of right, power, authority, responsibility for contributing to the definition of a problem, and how much people are affected by the solution.

By collaborating with an architectural firm, we recently encountered a convincing example illustrating

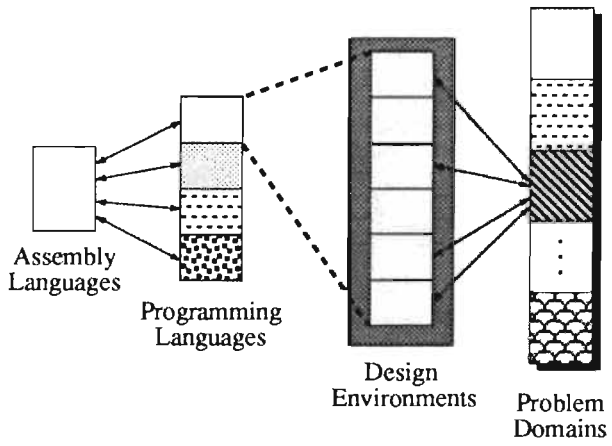

Figure 5: The 1990s: Domain-Oriented Design Environments

several of the issues discussed in this paper. The task the firm competed for was the design (and later construction) of the Denver Public Library. The final design competition took place in 1991, followed by construction to be finished by 1995. The City and County of Denver added as a constraint to the design that the library should not undergo major modifications for the first 15 years. While one may argue that the design task is to create a building, the real question is "what is the function of a public library for a large city in the year 2010?" Who is the owner of the problem? The client (i.e., the City and County of Denver, represented by librarians who have love affairs with books as well as techies who think there will be no books around in 20 years), the designers, and/or the customers who will eventually use the library?

In cases where is no single owner of a problem, the communication between all the "stakeholders" in the problem is of crucial importance. Shared understanding [Resnick 91] needs to be created. One way to do this is to develop languages of doing [Ehn 88] which create the possibility of mutual learning, thereby establishing a relationship of professional parity between system designers and system users.

How are Owners of Problem Put in Charge with Design Environments. The domain-orientation of design environments allows owners of problems to communicate with the systems at a level that is situated within their own world [Fischer, Lemke 88; Suchman 87; Wenger 90]. By supporting languages of doing [Ehn 88] such as prototypes, mock-ups, scenarios, created images, or visions of the future, design environments have the advantage of making it easier for owners of problems to participate in the design process, since the representations of the evolving artifacts are less abstract and less alienated from practical use situations. By keeping owners in the loop, they support the integration of problem setting and problem solving and allow software systems to deal with fluctuating and evolving requirements. By making information relevant to the task at hand [Fischer, Nakakoji 91], they are able to deliver the right knowledge, in the context of a problem or a service, at the right moment for a human professional to consider.

The Costs of NOT Putting Owners of Problems in Charge. By requiring high-tech scribes as intermediaries, designers are limited in solving ill-defined problems [Simon 73]. Ill-defined problems cannot be delegated because if they are delegated, situations do not "talk back" to the owners of the problems who

have the necessary knowledge to incrementally refine them. New requirements emerge during development, because they can not be identified until portions of the system have been designed or implemented. Traditional software design methodologies (such as the waterfall model, insisting on a strong separation between analysis and synthesis) have no chance to succeed in such situations. Alternatives approaches, such as methodologies allowing the co-evolution of specification and implementation, are needed [Swartout, Balzer 82; Fischer, Nakakoji 91].

Recreating Un-selfconscious Cultures of Design. Alexander [Alexander 64] has introduced the distinction between an un-selfconscious culture of design and a self-conscious culture of design. In an *un-selfconscious* culture of design, the failure or inadequacy of the form leads directly to an action to change or improve it (e.g., the owner of a house is its own builder, the form makers do not only make the form but they lived with it). This closeness of contact between designer and product allows constant rearrangement of unsatisfactory details. By putting owners of problems in charge, the positive elements of an un-selfconscious culture of design can be exploited in the development of software systems. Some of the obvious shortcomings of un-selfconscious culture of design, such as that they offer few opportunities for reflection is reduced by incorporating critics into the environments [Fischer et al. 91a].

5 Conclusions

Achieving the goal of putting problem owners in charge by developing design environments is not only a technical problem, but a considerable social effort. If the most important role for computation in the future is to provide people with a powerful medium for expression, then the medium should support them in working on the task, rather than requiring them to focus their intellectual resources on the medium itself.

The analogy to writing and its historical development suggest the goal "to take the control of computational media out of the hands of high-tech scribes." Pournelle (in BYTE, September 1990, p. 281 and p. 304) argues that "putting owners of problems in charge" has not always been the research direction of professional computer scientist: "*In Jesus' time, those who could read and write were in a different caste from those who could not. Nowadays, the high priesthood tries to take over the computing business. One of the biggest obstacles to the future of computing is C. C is the last attempt of the high priesthood to control the computing business. It's like the scribes and the Pharisees who did not want the masses to learn how to read and write.*"

Design environments are promising architectures to put owners of problems in charge. They are based on the basic belief that humans enjoy deciding and doing. They are based on the assumption that the experience of having participated in a problem makes a difference to those who are affected by the solution. People are more likely to like a solution if they have been involved in its generation; even though it might not make sense otherwise.

Acknowledgments

The author would like to thank the members of the Human-Computer Communication group at the University of Colorado who contributed to the conceptual framework and the systems discussed in this article. The research was supported by the National Science Foundation under grants No. CDA-8420944, IRI-8722792, and IRI-9015441; by the Army Research Institute under grant No. MDA903-86-C0143, and by grants from the Intelligent Interfaces Group at NYNEX, from Software Research Associates (SRA), Tokyo, and by the Software Designer's Associate (SDA) Consortium, Tokyo.

References

[Alexander 64]
C. Alexander, *The Synthesis of Form*, Harvard University Press, 1964.

[Belady 85]
L. Belady, *MCC: Planning the Revolution in Software*, IEEE Software, November 1985, pp. 68-73.

[Brooks 87]
F.P. Brooks Jr., *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

[CSTB 90]
Computer Science and Technology Board, *Scaling Up: A Research Agenda for Software Engineering*, Communications of the ACM, Vol. 33, No. 3, March 1990, pp. 281-293.

[Curtis, Krasner, Iscoe 88]
B. Curtis, H. Krasner, N. Iscoe, *A Field Study of the Software Design Process for Large Systems*, Communications of the ACM, Vol. 31, No. 11, November 1988, pp. 1268-1287.

[Denning 88]
P. Denning, *Awakening*, Communications of the ACM, Vol. 31, No. 11, November 1988, pp. 1254-1255.

[Ehn 88]
P. Ehn, *Work-Oriented Design of Computer Artifacts*, Almquist & Wiksell International, 1988.

[Fischer et al. 91a]
G. Fischer, A.C. Lemke, R. McCall, A. Morch, *Making Argumentation Serve Design*, Human Computer Interaction, Vol. 6, No. 3-4, 1991, pp. 393-419.

[Fischer et al. 91b]
G. Fischer, A.C. Lemke, T. Mastaglio, A.I. Morch, *Critics: An Emerging Approach to Knowledge-Based Human Computer Interaction*, International Journal of Man-Machine Studies, Vol. 35, No. 5, 1991, pp. 695-721.

[Fischer, Girgensohn 90]
G. Fischer, A. Girgensohn, *End-User Modifiability in Design Environments*, Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA), ACM, New York, April 1990, pp. 183-191.

[Fischer, Lemke 88]
G. Fischer, A.C. Lemke, *Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication*, Human-Computer Interaction, Vol. 3, No. 3, 1988, pp. 179-222.

[Fischer, McCall, Morch 89]
G. Fischer, R. McCall, A. Morch, *Design Environments for Constructive and Argumentative Design*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 269-275.

[Fischer, Nakakoji 91]
G. Fischer, K. Nakakoji, *Making Design Objects Relevant to the Task at Hand*, Proceedings of AAAI-91, Ninth National Conference on Artificial Intelligence, AAAI Press/The MIT Press, Cambridge, MA, 1991, pp. 67-73.

[Fischer, Rathke 88]
G. Fischer, C. Rathke, *Knowledge-Based Spreadsheet Systems*, Proceedings of AAAI-88, Seventh National Conference on Artificial Intelligence (St. Paul, MN), Morgan Kaufmann Publishers, San Mateo, CA, August 1988, pp. 802-807.

[Fischer, Reeves 92]
G. Fischer, B.N. Reeves, *Beyond Intelligent Interfaces: Exploring, Analyzing and Creating Success Models of Cooperative Problem Solving*, Applied Intelligence, Special Issue Intelligent Interfaces, 1992, (in press).

[Henderson, Kyng 91]
A. Henderson, M. Kyng, *There's No Place Like Home: Continuing Design in Use*, in J. Greenbaum, M. Kyng (eds.), *Design at Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1991, pp. 219-240, ch. 11.

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

[Lave 88]
J. Lave, *Cognition in Practice*, Cambridge University Press, Cambridge, UK, 1988.

[Lemke, Fischer 90]
A.C. Lemke, G. Fischer, *A Cooperative Problem Solving System for User Interface Design*, Proceedings of AAAI-90, Eighth National Conference on Artificial Intelligence, AAAI Press/The MIT Press, Cambridge, MA, August 1990, pp. 479-484.

[Polanyi 66]
M. Polanyi, *The Tacit Dimension*, Doubleday, Garden City, NY, 1966.

[Resnick 91]
L.B. Resnick, *Shared Cognition: Thinking as Social Practice*, in L.B. Resnick, J.M. Levine, S.D. Teasley (eds.), *Perspectives on Socially Shared Cognition*, American Psychological Association, Washington, D.C., 1991, pp. 1-20, ch. 1.

[Rittel 84]
H.W.J. Rittel, *Second-Generation Design Methods*, in N. Cross (ed.), *Developments in Design Methodology*, John Wiley & Sons, New York, 1984, pp. 317-327.

[Schoen 83]
D.A. Schoen, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.

[Shaw 89]
M. Shaw, *Maybe Your Next Programming Language Shouldn't Be a Programming Language*, in Computer Science and Technology Board (eds.), *Scaling Up: A Research Agenda for Software Engineering*, National Academy Press, 1989, pp. 75-82.

[Sheil 83]
B.A. Sheil, *Power Tools for Programmers*, Datamation, February 1983, pp. 131-143.

[Simon 73]
H.A. Simon, *The Structure of Ill-Structured Problems*, Artificial Intelligence, No. 4, 1973, pp. 181-200.

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

[Stefik 86]
M.J. Stefik, *The Next Knowledge Medium*, AI Magazine, Vol. 7, No. 1, Spring 1986, pp. 34-46.

[Suchman 87]
L.A. Suchman, *Plans and Situated Actions*, Cambridge University Press, Cambridge, UK, 1987.

[Swartout, Balzer 82]
W.R. Swartout, R. Balzer, *On the Inevitable Intertwining of Specification and Implementation*, Communications of the ACM, Vol. 25, No. 7, July 1982, pp. 438-439.

[Wenger 90]
E. Wenger, *Toward a Theory of Cultural Transparency: Elements of a Social Discourse of the Visible and the Invisible*, Dissertation, Information and Computer Science, University of California, Irvine, CA, 1990.

[Winograd 79]
T. Winograd, *Beyond Programming Languages*, Communications of the ACM, Vol. 22, No. 7, July 1979, pp. 391-401.

[Winograd, Flores 86]
T. Winograd, F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ, 1986.