The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design

Gerhard Fischer
University of Colorado, Center for LifeLong Learning and Design (L3D)
Department of Computer Science, Campus Box 430
Boulder, CO 80309-0430 - USA
gerhard@cs.colorado.edu

ABSTRACT

Complex (software) design problems require more knowledge than any single person or any single group possesses because the knowledge relevant to a problem is distributed among many different stakeholders. Software reuse exploits a collaboration process in which designers working on new problems can take advantage of the work of designers who have encountered similar problems in the past. Not only technical problems but also *cognitive* and *social* factors inhibit the widespread success of systematic software reuse. An important paradigm shift is to reconceptualize reuse as a *collaborative* process, in which software designers should not only take advantage of existing reuse repositories, but, through their own work, modify components and evolve reuse repositories. We discuss *conceptual frameworks, practices,* and *systems* that support software design as a collaborative knowledge construction process.

KEYWORDS

software development, reuse repositories, system evolution, reuse, collaborative software construction, open source, communities of practice, communities of interest, human aspects of computer applications

SOFTWARE REUSE: PROMISES AND CHALLENGES

The basic premise of software reuse is support for design methodologies for which the main activity is not the building of new systems from scratch, but the integration, modification, and explanation of existing ones [Winograd, 1996]. Software reuse is a promising design methodology because complex systems develop faster if they can be built on *stable subsystems* [Simon, 1996] and because reuse supports *evolution* [Dawkins, 1987]. Various types of software artifacts, such as design knowledge, application domain knowledge, software architectures, design patterns, and components, can be reused. However, merely providing a reuse repository is not enough.

Reuse is not only a technical problem, it is also a cognitive and social problem [Fischer, 1987]. Designers must locate reusable software artifacts relevant to their tasks and understand them. In addition, *new social practices and tools* are needed to encourage and support designers to contribute to the reuse repository.

Table 1 contrasts some of the past and present concerns of creating complex software systems that software technologies of the future must address.

Table 1: Past and Present Concerns for Software Technologies

dimension	past	present
limiting resource	information	human attention
models for collaboration	access	informed participation
design tools	focus: "downstream activities" — robust implementations of given specifications	focus: "upstream activities" — co-evolution between problem framing and problem solving
design products	finished systems	evolution
support for collaboration	file transfer	world-wide web (WWW)
model for creation	individual creativity	social creativity
documents	formal and informal objects of specific communities of practices	boundary objects: supporting collaboration between different communities
focus of software reuse	technical issues	cognitive, social issues
intellectual property	closed, company-owned	models for sharing (e.g., open source)

THE LOCATION, COMPREHENSION, AND MODIFICATION CYCLE

The use of reusable objects suffers from the problem of information overload. Developers do not know what reusable artifacts exist; how to access them; how to understand them; and/or how to combine, adapt, and modify them to meet current needs. These challenges exist in each phase of the *location-comprehension-*

modification cycle [Fischer et al., 1991] depicted in Figure 1. Designers first have to locate potentially useful pieces of information (either through access mechanisms or delivery mechanisms), comprehend the retrieved information, and modify it according to their current needs. We have developed three systems to support the *location-comprehension-modification cycle*: Codefinder, Explainer, and Modifier.

Location. Codefinder [Henninger, 1993] supports the process of retrieving software objects when information needs are ill-defined and users are not familiar with the vocabulary [Furnas et al., 1987] of the repository through an innovative integration of retrieval by reformulation, and spreading activation.

Comprehension. Explainer [Redmiles, 1992] supports programmers' use of examples as a powerful aid to problem solving. Examples not only provide objects to be reused but also present a context in which users can explore issues related to the task-at-hand.

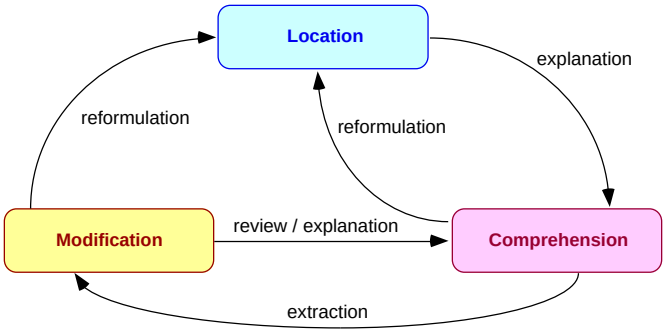

Figure 1: The Location-Comprehension-Modification Cycle

Modification. Modifier [Girgensohn, 1992] supports designers, specifically *end-users* and *local developers* (some end-users who have learned enough about a systems to make some modifications) [Nardi, 1993], in adapting reusable artifacts to their needs. Modification is necessary because different designers pursue different tasks, have different preferences, and have evolving needs or requirements due to changes in the world.

CREATING SHARED UNDERSTANDING BETWEEN DIFFERENT DESIGN COMMUNITIES

System development is difficult because it requires creating a shared understanding among different design communities. Over the years, we have developed a number of conceptual frameworks to facilitate the creation of shared understanding among different design communities.

Supporting Human Problem-Domain Communication with Domain-Oriented Systems. Domain-oriented systems avoid the pitfall of excess generality. Instead of serving all needs obscurely and insufficiently with general-purpose programming languages, domain-oriented systems serve a few needs well. The semantics of computing environments need to be better tuned to specific domains matching the mental models of the users. Human-computer communication needs to be advanced to *human-problem domain communication*, where the computer becomes "*invisible*" and users have the feeling of direct interaction with a problem domain.

Situation and System Models. When software designers approach a problem, they often begin at a high level of abstraction, and conceptualize the design in terms of the application problem to be solved [Curtis et al., 1988]. This initial conceptualization must then be translated into terms and abstractions that the computer can understand. The gap between the application level and system level in conventional software engineering environments is large. The underlying problem can be characterized as a mismatch between the *system model* provided by the computing environment and the *situation model* of the user [Kintsch, 1998].

Putting Owners of Problems in Charge. Problems that can be clearly defined can be delegated. If a complete problem description could exist apart from its solution, then it would be possible to "delegate" that problem description to an intermediary. Compared to problem owners, however, intermediaries are severely limited when acting on an ill-defined problem. A key attribute of a problem is that the owner has the authority to change its description. The difficulty with delegating ill-defined problems is that the owner of the problem interacts only indirectly with the emergent solution and thus is not able to foresee implications that certain specifications and assumptions may have on the final solution.

SUPPORTING SOFTWARE SYSTEMS AS LIVING ENTITIES: THE SEEDING, EVOLUTIONARY GROWTH, RESEEDING PROCESS MODEL

We live in a world characterized by *evolution*. Biology tells us that complex, natural systems are not created all at once but must instead evolve over time. We are becoming increasingly aware that evolutionary processes are ubiquitous and critical for technological innovations as well. This is particularly true for complex software systems because these systems do not necessarily exist in a technological context alone but instead are embedded within dynamic human organizations. For many problems, software design is best understood as an evolutionary process in which system requirements and functionality are determined through an iterative process of collaboration among multiple stakeholders, rather than being completely specified before system development occurs [Curtis et al., 1988]. Our research focuses on the following claims about software systems embedded within dynamic human organizations: (1) they must evolve because they cannot be completely designed prior to use, (2) they must evolve to some extent at the hands of the users, and (3) they must be designed for evolution.

The *Seeding, Evolutionary Growth, Reseeding (SER)*

model [Fischer et al., 2001] is a process model that describes the evolution of complex systems, including the development of operating systems, design environments, reuse repositories, and open source development efforts [Raymond & Young, 2001].

Seeding. In the seeding phase, system developers and users work together to develop an initial *seed*. As the name suggests, the seed is considered as a starting point for ongoing growth. System developers are necessary in the seeding phase because the product is a complex software system. Participation of users is also necessary because they have the knowledge necessary to decide what content should be included in the seed, and how the content will need to evolve over time. Although the SER model acknowledges that the initial seed cannot be complete, the seeding process still requires a substantial up-front investment.

Evolutionary Growth. During this phase, the information repository plays two simultaneous roles: (1) through dissemination, it informs work; and (2) through integration, it accumulates the products of work. An essential aspect of this phase is that the user community is responsible for making changes to the seed. Making contributions of domain knowledge should be a part of everyone's job. But formalization of information and modification of system functionality may require significant programming knowledge, and therefore will be the responsibility of *local developers* [Nardi, 1993] who are technically inclined and motivated to do this work. During the evolutionary growth phase, the software designers are not present. Therefore, it is necessary to allow some new design knowledge to be added by the users, thus requiring computational mechanisms that support end-user modifiability [Girgensohn, 1992] and end-user programming [Repenning et al., 1998].

Reseeding. From the perspective of software systems, evolutionary growth increases the chaos of the original system that makes further growth impossible. Reseeding is a process to reduce such chaos. Reseeding is a complex process by which users, together with system developers, must take a stake in the current system, synthesize the current state of the system, and reconceptualize the system. The result of the reseeding process is a new system that can serve as the basis for future evolution. The cycle of evolution and reseeding continues as long as people actively use the system to solve problems.

EXAMPLES OF SYSTEMS

Over the last decade, our conceptual frameworks co-evolved with system building efforts to make reuse more successful and move it from a one-way street of knowledge transfer to a collaborative knowledge construction process. Table 2 gives an overview; Codefinder, Explainer, and Modifier were briefly discussed previously, and three other system building efforts are described in this section.

Table 2: Overview of Conceptual Frameworks and Systems

Fundamental Challenge	Conceptual Frameworks	Systems
Complex systems; high-functionality applications	Software Reuse	Codefinder, Explainer, Modifier
Problem-specific interaction	Human-Problem Domain Interaction; Domain-Oriented Design Environments	KID (Knowing-in-Design)
Organization of large bodies of knowledge; information overload	Personalized Information Delivery	CodeBroker
Collaboration	Social Creativity	Envisionment and Discovery Collaboratory

Domain-Oriented Design Environments

Domain-oriented design environments (DODEs) [Fischer, 1994] reduces the large conceptual distance between problem-domain semantics and software artifacts. The integration among different components of DODEs supports the co-evolution of specification and construction while allowing designers to access relevant knowledge at each stage within the software development process. DODEs have been used for the design of such software artifacts as user interfaces, voice dialog systems and Cobol programs, and have served equally well for the conceptual design of such material artifacts as kitchens, lunar habitats, and computer networks. The fundamental assumption behind our research is that DODEs will become as valuable and as ubiquitous in the future as compilers have been in the past.

An important aspect of DODEs is making information relevant to the task-at-hand. The scarce resource in our information-rich society is attention. To address the problem of information overload, environments must focus on providing workers with the information they need, and at a time when they need it. The standard approach for knowledge dissemination is to support *access* with either search or browsing. Although such approaches are necessary to locate information, they are not sufficient for the following reasons: (1) users may not be able to articulate their information needs in a way that the access mechanisms require; (2) users may not be motivated to search for information if they are not aware of the existence of information relevant to their needs; and (3) users may not be aware of a need for information in the first place.

Figure 2 shows the KID (Knowing-in-Design) [Nakakoji, 1993] system, a DODE for the domain of kitchen design. KID increases the chance that designers will encounter useful design knowledge as they design. Design

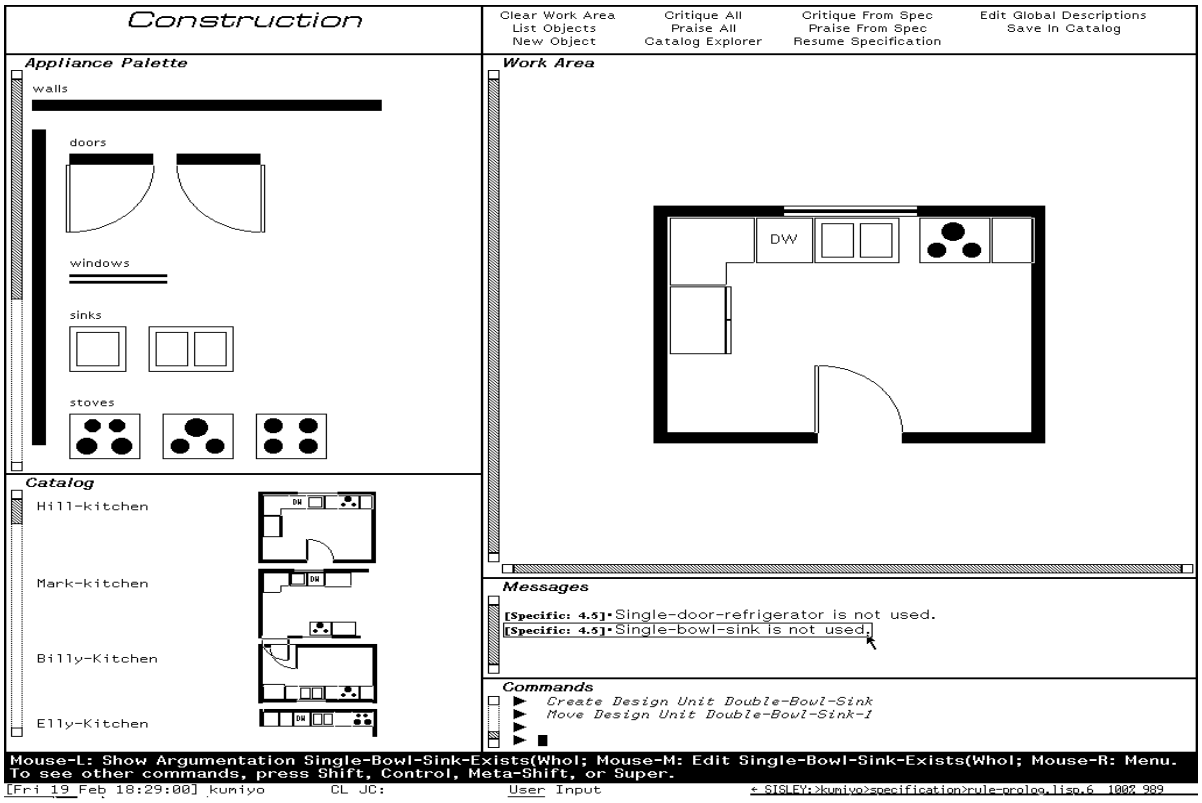

Figure 2: KID (Knowing-in-Design) — A DODE for Kitchen Design

knowledge stored in KID is integrated, so different types of design knowledge are linked together. *Critics* in KID [Fischer et al., 1998] check the compliance of the current construction to the current specification supporting designers in the co-evolution of a partial specification (problem) and a partial construction (solution).

The CodeBroker System

The long-term goal of software reuse is thwarted by an inherent design conflict: to be useful, a reuse system must provide many building blocks (thereby facing all design challenges associated with *high-functionality applications* [Fischer, 2001b]), but when many building blocks are available, finding and choosing an appropriate one (one relevant to the task-at-hand) becomes a difficult problem.

CodeBroker [Ye, 2001] (see Figure 3) is a system that supports software reuse by *delivering* components relevant to the task-at-hand. The *task-at-hand* is inferred from a partially written program by exploiting information contained in comments and signatures. It identifies the most promising components in the repository and delivers them to the designer. To avoid confronting designers with too much information that they may already know or in which they have indicated no interest, user and discourse models are employed to make the information not only relevant to the task-at-hand, but personalized to the goals and background knowledge of specific designers.

The Envisionment and Discovery Collaboratory

The Envisionment and Discovery Collaboratory (EDC) [Arias et al., 2000] supports social creativity by empowering stakeholders to act as designers, allowing them to create shared understanding, to contextualize information to the task-at-hand, and to create boundary objects (objects that are understood by designers coming from different communities of practice) [Fischer, 2001a] in collaborative design activities. The EDC framework is applicable to different domains. Figure 4 shows part of the current prototype of the EDC (which explored urban transportation planning as an application domain). The EDC attempts to *leverage* the powerful collaboration that can take place in face-to-face settings and *augment* this collaboration with boundary objects by integrating our previous research work in simulation, decision-support, and domain-oriented design environments.

The vision behind the EDC is to provide contextualized support for reflection-in-action [Schön, 1983] within collaborative design activities. Using the horizontal electronic whiteboard, participants work "around the table" incrementally creating a shared model of the problem. They interact with computer simulations by manipulating three-dimensional, physical objects, which constitute a language for the domain. The position and movement of these physical objects are recognized by means of the touch-sensitive projection surface. The vertical board provides information for reflection about

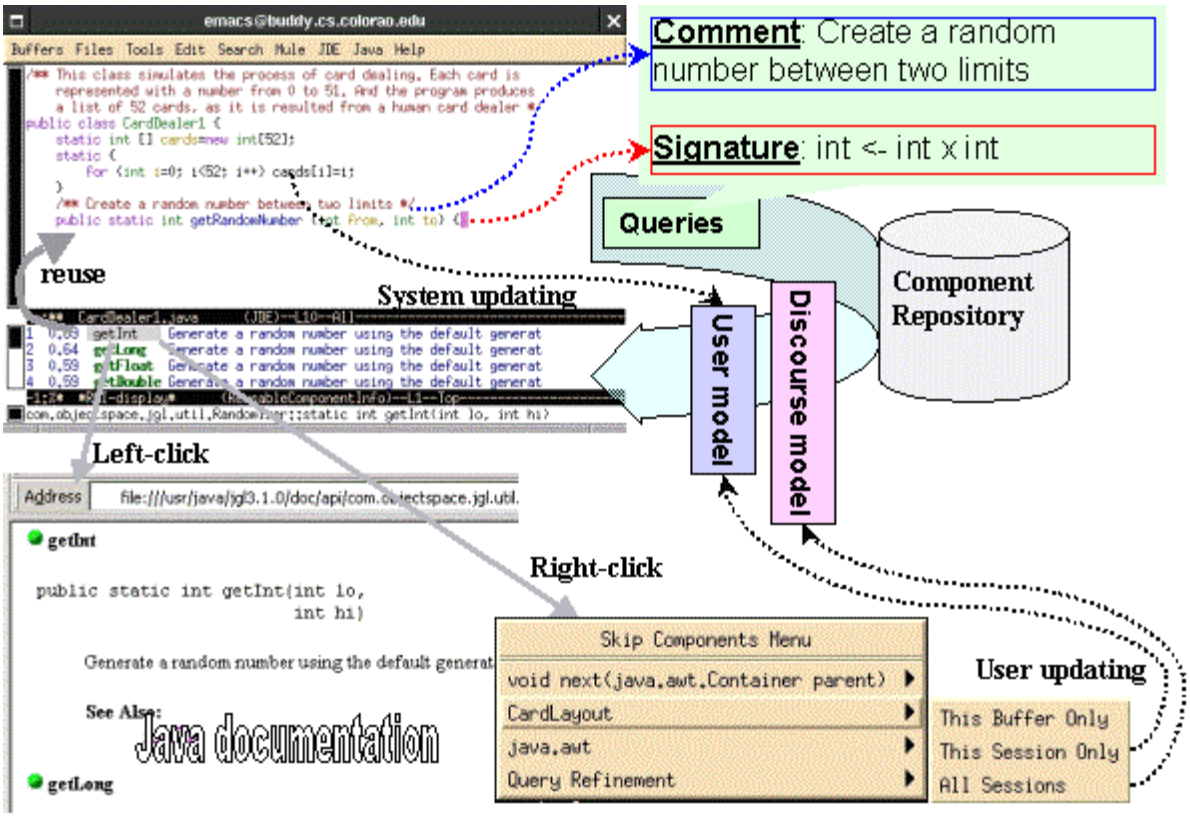

Figure 3: The CodeBroker System

the events occurring in the action space.

CHALLENGES FOR THE FUTURE

Understanding Promises and Pitfalls

Taking Evolutionary Models too Literally. We are convinced that models from biology will be more relevant to future software systems than models from mathematics. But we have to be cautious: to follow an evolutionary approach in software design *successfully* does not imply that concepts from biological evolution should be mimicked literally, but rather that they need to be reinterpreted in the domain of software design [Basalla, 1988] — an attempt we undertake with the SER model.

Beyond Information. It is a myth that *"anytime and anywhere"* access to information will solve the information overload problem for designers. The scarce resource for most people is *attention*: the real challenge is to *"say the 'right' thing at the 'right' time in the 'right' way"* [Fischer, 2001b]. This can be done only with computational environments that are able to take the user's context into account (e.g., what the users are doing, what they know, where they are, what they have done in the past). Future software technologies need to support *attention economies* [Brown & Duguid, 2000], in which attention is the most valued resource.

End-User Modification and Programming for Communities: Evolution at the Hands of Users. Because end-users experience breakdowns and insufficiencies of software systems in their work, *they* should be able to report, react to, and resolve those problems. At the core of our approach to evolutionary design lies the ability of end-users (in our case, domain designers) to make significant changes to system functionality and to share those modifications within a community of designers. Mechanisms for end-user modification and programming are, therefore, a cornerstone of evolvable systems. DODEs (and specifically components such as Modifier) make end-user modifications feasible because they support human problem-domain interaction. We do not assume that all domain designers will be willing or interested in making system changes, but within local communities of practice *local developers* often exist who are interested in and capable of performing these tasks.

Beyond Technologies: Socio-Technical Environments. An important nontechnical challenge for collaborative construction and evolution of information repositories is to take *motivation* seriously. For sustained collaborative work practices, an incentive must exist to create *social capital* [Raymond & Young, 2001] by rewarding stakeholders for contributing and receiving knowledge as a member of a community. Social capital can be characterized as follows: (1) human beings have an innate drive to compete for social status; (2) social status is

Figure 4: The Current Prototype of the EDC

determined not by what you control but by *what you give away*; (3) prestige is a good way to attract attention and cooperation from others; and (4) utilization is the sincerest form of flattery.

From Reuse to Collaborative Knowledge Construction

Sustained Knowledge Creation. The basic assumption underlying our transitions from reuse to collaborative knowledge construction is that knowledge is not a commodity to be consumed but is collaboratively designed and constructed. This focus emphasizes innovation, continuous learning, and collaboration as important processes. Collaborative design not only requires building on knowledge as information stored in repositories; it is a continual process in which knowledge is created as a by-product of work, integrated in an open and evolving repository, and then disseminated to others in the organization when it is relevant to their work.

Systems must undergo sustained development, requiring extensible systems and social structures that include users who are able to make changes in systems [Henderson & Kyng, 1991; Nardi, 1993]). These requirements are both technical and social. Promising technical approaches to enable continual evolution of systems include end-user modification [Girgensohn, 1992] and end-user programming [Repenning et al., 1998]. Development practices and products must acknowledge the importance of preserving knowledge that was generated in the design process and it should be made part of the system. A key issue for the capture of process information is the effort required in addition to the normal development tasks [Grudin, 1994].

Collaborative Design. Increasingly, design tasks are done not by individuals, but by groups or communities working together. Complexity in collaborative design arises from the need to synthesize different perspectives of a problem, the management of large amounts of information relevant to a design task, and the understanding of the design decisions that have determined the long-term evolution of a designed artifact.

Our work has focused on two types of groups, *communities of practice* (CoPs) [Wenger, 1998] and *communities of interest* (CoIs) [Fischer, 2001a]:

1. CoPs consist of people sharing a common practice, or domain of interest (e.g., software engineers). CoPs are sustained over time. They provide a means for newcomers to learn about the practice and for established members to share knowledge about their work and to collaborate on projects. CoPs need support to understand the long-term evolution of artifacts as well as the problems caused by rapid change in their domain.

2. CoIs consist of people from *different* fields who come together to work on a particular project or problem (e.g., teams involved in software development, especially in the requirement analysis phase, are best understood as CoIs involving at least software engineers and users). CoIs typically exist for the duration of the project. They need support for creating shared understanding among stakeholders from different backgrounds, who bring different perspectives and languages to the problem.

CoIs are important for software design as projects become more interdisciplinary and collaborative design brings together specialists from many domains. System developments such as the EDC try to integrate multiple perspectives and help CoIs to create shared understanding. The EDC supports "design as a conversation with external representations" [Schön, 1983] that are used not only to facilitate a conversation with the design situation, but with other designers as well.

CONCLUSIONS

The development of complex software systems is challenging design activity. The process is made difficult *"not because of the complexity of technical problems, but because of the social interaction when users and system developers learn to create, develop and express their ideas and visions"* [Greenbaum & Kyng, 1991]. Designing complex software systems is an intrinsically collaborative process in which the major source of complexity arises from the need to synthesize different perspectives on the problems to be solved. These perspectives originate from the many stakeholders involved in system development. The fundamental challenge for software technologies of the future is to provide support for achieving a shared understanding among groups of people that see the world in fundamentally different ways.

Acknowledgements

The author thanks the members of the Center for LifeLong Learning and Design at the University of Colorado, who have made major contributions to the conceptual frameworks described in this paper. A special thanks goes to the authors of the systems used as examples in this paper: Scott Henninger (Codefinder); David Redmiles (Explainer); Andreas Girgensohn (Modifier); Kumiyo Nakakoji (KID); Yunwen Ye (CodeBroker); and Ernesto Arias, Hal Eden, Andy Gorman, and Eric Scharff (EDC). The feedback from Yunwen Ye and Taro Adachi on an earlier version of this paper led to numerous improvements.

The research was supported by (1) the National Science Foundation, Grant REC-0106976; (2) SRA Key Technology Laboratory, Tokyo, Japan; and (3) the Coleman Family Foundation, San Jose, CA.

REFERENCES

Arias, E. G., Eden, H., Fischer, G., Gorman, A., & Scharff, E. (2000) "Transcending the Individual Human Mind— Creating Shared Understanding through Collaborative Design," *ACM Transactions on Computer Human-Interaction,* 7(1), pp. 84-113.

Basalla, G. (1988) *The Evolution of Technology,* Cambridge University Press, New York.

Brown, J. S. & Duguid, P. (2000) *The Social Life of Information,* Harvard Business School Press, Boston, MA.

Curtis, B., Krasner, H., & Iscoe, N. (1988) "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM,* 31(11), pp. 1268-1287.

Dawkins, R. (1987) *The Blind Watchmaker,* W.W. Norton and Company, New York - London.

Fischer, G. (1987) "Cognitive View of Reuse and Redesign," *IEEE Software, Special Issue on Reusability,* 4(4), pp. 60-72.

Fischer, G. (1994) "Domain-Oriented Design Environments," *Automated Software Engineering,* 1(2), pp. 177-203.

Fischer, G. (2001a) "Communities of Interest: Learning through the Interaction of Multiple Knowledge Systems," *24th Annual Information Systems Research Seminar In Scandinavia (IRIS'24), Ulvik, Norway*, pp. 1-14.

Fischer, G. (2001b) "User Modeling in Human-Computer Interaction," *User Modeling and User-Adapted Interaction (UMUAI), Dordrecht, The Netherlands: Kluwer Academic Publishers,* 11(2), pp. 65-86.

Fischer, G., Grudin, J., McCall, R., Ostwald, J., Redmiles, D., Reeves, B., & Shipman, F. (2001) "Seeding, Evolutionary Growth and Reseeding: The Incremental Development of Collaborative Design Environments." In G. M. Olson, T. W. Malone, & J. B. Smith (Eds.), *Coordination Theory and Collaboration Technology,* Lawrence Erlbaum Associates, Mahwah, New Jersey, pp. 447-472.

Fischer, G., Henninger, S. R., & Redmiles, D. F. (1991) "Cognitive Tools for Locating and Comprehending Software Objects for Reuse." In *Thirteenth International Conference on Software Engineering (Austin, TX),* IEEE Computer Society Press, Los Alamitos, CA, pp. 318-328.

Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., & Sumner, T. (1998) "Embedding Critics in Design Environments." In M. T. Maybury & W. Wahlster (Eds.), *Readings in Intelligent User Interfaces,* Morgan Kaufmann, San Francisco, pp. 537-561.

Furnas, G. W., Landauer, T. K., Gomez, L. M., & Dumais, S. T. (1987) "The Vocabulary Problem in Human-System Communication," *Communications of the ACM,* 30(11), pp. 964-971.

Girgensohn, A. (1992) *End-User Modifiability in*

Knowledge-Based Design Environments, Ph.D. Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

Greenbaum, J. & Kyng, M. (Eds.) (1991) *Design at Work: Cooperative Design of Computer Systems,* Lawrence Erlbaum Associates, Inc., Hillsdale, NJ.

Grudin, J. (1994) "Groupware and social dynamics: Eight challenges for developers," *Communications of the ACM,* 37(1), pp. 92-105.

Henderson, A. & Kyng, M. (1991) "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, Inc., Hillsdale, NJ, pp. 219-240.

Henninger, S. R. (1993) *Locating Relevant Examples for Example-Based Software Design*, Ph. D Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

Kintsch, W. (1998) *Comprehension: A Paradigm for Cognition,* Cambridge University Press, Cambridge, England.

Nakakoji, K. (1993) *Increasing Shared Understanding of a Design Task Between Designers and Design Environments: The Role of a Specification Component*, Ph.D. Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

Nardi, B. A. (1993) *A Small Matter of Programming,* The MIT Press, Cambridge, MA.

Raymond, E. S. & Young, B. (2001) *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary,* O'Reilly & Associates, Sebastopol, CA.

Redmiles, D. F. (1992) *From Programming Tasks to Solutions: Bridging the Gap Through the Explanation of Examples*, Ph.D. Dissertation, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

Repenning, A., Ioannidou, A., Rausch, M., & Phillips, J. (1998) "Using Agents as a Currency of Exchange between End-Users," *Proceedings of the WebNET 98 World Conference of the WW, Internet, and Intranet*, pp. 762-767.

Schön, D. A. (1983) *The Reflective Practitioner: How Professionals Think in Action,* Basic Books, New York.

Simon, H. A. (1996) *The Sciences of the Artificial,* (Third ed.), The MIT Press, Cambridge, MA.

Wenger, E. (1998) *Communities of Practice — Learning, Meaning, and Identity,* Cambridge University Press, Cambridge, England.

Winograd, T. (Ed.) (1996) *Bringing Design to Software,* ACM Press and Addison-Wesley, New York.

Ye, Y. (2001) *Supporting Component-Based Software Development with Active Component Repository Systems*, Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO.