Contribution to the Special Issue of the Automated Software Engineering Journal on *"Reflections on Automated Software Engineering"*

Rethinking Software Design in Participation Cultures

Gerhard Fischer
Center for Lifelong Learning and Design
University of Colorado, Boulder
gerhard@colorado.edu

Abstract

The research activities in software engineering at the Center for LifeLong Learning & Design (L3D) in the past have been grounded in the basic assumption that important aspects of software engineering are best understood as *human-centered design* activities. Some of the major objectives were to support designers with domain-oriented design environments, allowing them to interact at the problem domain level and to frame activities and artifacts based on an evolutionary approach.

A fundamental shift occurring over the last few years is the formation of *participation cultures* enhanced and supported by a change from an *industrialized information economy* (specialized in producing finished goods to be consumed passively) to a cyber-enabled *networked information economy* (in which all people are provided with the means to participate actively in personally meaningful problems). Some of the implications of this fundamental shift for software engineering, including meta-design, lessons learned from open source software, and distribution and diversity in communities, are explored, and their implications for the "automate/informate" perspectives are briefly discussed.

Keywords

software design; domain-oriented design environments; human-problem domain interaction; meta-design; distribution and diversity; networked information economy; participation cultures

Introduction

The Center for LifeLong Learning & Design (L3D) at the University of Colorado has been involved in research on software engineering for several decades. We have explored software engineering from a number of different perspectives and have created tools and environments to support a variety of different approaches, including:

- a *domain-oriented* perspective supported by domain-oriented design environments [Fischer, 1994];
- a *reuse and redesign* perspective facilitated by tools in support of location, comprehension, modification, and sharing [Ye & Fischer, 2005];
- an *evolutionary* perspective supported by the seeding, evolutionary growth, reseeding model [Fischer, 1998]; and
- a *collaboration* perspective facilitated by meta-design and supported by participation cultures [Fischer & Giaccardi, 2006].

These perspectives were not pursued independently of each other but were explored as different facets of a human-centered design approach to software engineering [Fischer, 2003; Winograd, 1996]. In doing so, our work involved *interdisciplinary relationships* of software engineering with human-computer interaction (HCI), design of interactive systems (DIS), participatory design (PD), computer-supported collaborative learning (CSCL), and computer-supported cooperative work (CSCW). Our research was inspired by ideas that were developed by designers in other fields, including *convivial tools* [Illich, 1973], *science of design* [Simon, 1996], *design patterns* [Alexander et al., 1977], *reflection-in-action and critiquing* [Schön, 1983], *evolutionary models* [Dawkins, 1987], and *user-driven innovation* [von Hippel, 2005].

Our work was grounded in the basic belief that system development is difficult not because of the complexity of technical problems, but because of the social interaction between users and system developers as they learn to create, develop, and express their ideas and visions [Greenbaum & Kyng, 1991]. Such social interaction should be pursued and supported with socio-technical environments [Mumford, 1987].

A Design- and Human-Centered Perspective of Software Engineering

Traditional software engineering research has been primarily concerned with the transition from specifications to implementations (*"downstream" activities*) rather than with the problem of how faithfully specifications really address the problems to be solved (*"upstream" activities*) [Belady & Lehman, 1985] (see Figure 1).

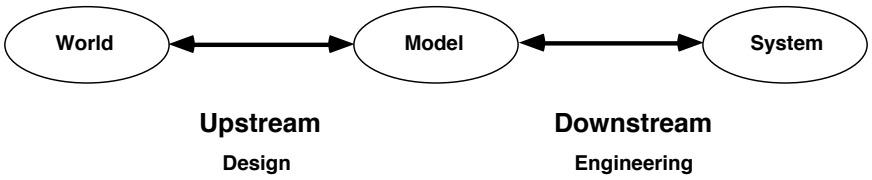

Figure 1: Upstream and Downstream Activities

Many methodologies and technologies were developed to prevent *implementation disasters* by producing correct programs with respect to a given specification. The progress made to successfully reduce implementation disasters, such as structured programming and information hiding, allowed an equally relevant problem to surface: how to prevent design disasters [Lee, 1992]. *Design disasters* refer to a situation in which a correct implementation with respect to a given specification is of little value because the specification does not adequately address the problem to be solved.

Table 1 summarizes comparisons between downstream and upstream activities. Paying more attention to upstream activities is not a rejection of approaches and methods that are more formal-based, but a shift of the primary point of view: software systems definitely need to be

correct and efficient, but what is the value of these systems, if they are not relevant, suitable, engaging, and adequate to users and their needs

Table 1: Comparison between Upstream and Downstream Activities

	Upstream	Downstream
type of problem	ill-defined problems	well-defined problems
breakdowns	design disasters (wrong problem is solved)	implementation disasters (wrong solution to the right problem)
focus	embedding in larger context, user experience	computational mechanisms
primary source of knowledge	owners of problems, domain workers	software engineers, programmers
support environments	domain-oriented design environments (DODEs)	knowledge-based software assistants, programming and testing environments
interaction paradigm	languages of doing: prototypes, scenarios, mock-ups, boundary objects	(formal) specifications
externalization	(semi-formal) objects-to-think-with understood by all stakeholders	computationally interpretable objects
criteria to judge solutions	adequate, understandable, enjoyable, engaging	correct, robust, reliable, meets functional specifications
evolution	meta-design, end-user development, users	debugging, verification, validation

Domain-Oriented Design Environments

Orienting software systems toward specific domains or tasks has been heralded by many researchers as a means of making software both more useful and more usable [Shaw, 1989]. Domain-oriented software is more usable than generic software because developers and users being knowledge workers in a specific domain are able to directly interact with familiar entities and do not need to learn unfamiliar computer-specific concepts.

When computer systems first emerged, users were required to express themselves in the machine language of that system. These languages were completely general: their semantics was not tied to any specific problem domain. The conceptual distance for a human (working in a certain domain) who wanted to model a problem was very large. The first fundamental development was the creation of assembly languages and high-level programming languages (see Figure 2). These developments still retained the generality, but they facilitated and supported specific domain-oriented operations (e.g., matrices in APL, lists and trees in Lisp, and objects in Smalltalk). At the same time, these developments lead to a *division of labor* [Levy & Murnane, 2004]: compiler developers emerged as a new class of computer professionals who developed compilers, thereby allowing most of the professional programmers to program in higher-level languages. The initial developments represented the initial step toward creating computational environments to support users engaged in specific domains [Fischer, 1994], thereby supporting not only human-computer interaction but *human problem-domain interaction* [Fischer & Lemke, 1988].

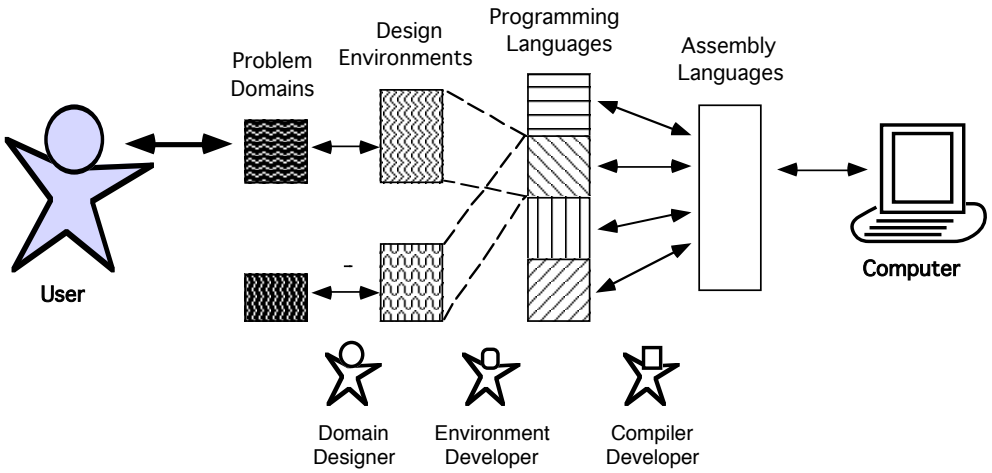

Figure 2: A Layered Architecture in Support of Human Problem-Domain Interaction

These developments illustrate the fundamental design tradeoff between generality and specificity. Generality is a highly desirable goal because the same tool can be used in many different contexts. However, tools that are broadly applicable for all kinds of users and tasks come with a substantial cost, which can be characterized by the Turing Tar Pit [Perlis, 1982]: *"Beware of the Turing Tar Pit, in which everything is possible, but nothing of interest is easy."* These environments are based on a level of representation that is too far removed from the conceptual world of knowledge workers in specific domains. They emphasize *objective computability* (i.e., what can be computed in principle), but they pay little attention to *subjective computability* (i.e., what can people do with a reasonable amount of effort and with limited knowledge about the computational environment).

The other end of the design spectrum can be characterized by the Inverse of the Turing Tar Pit: *"Beware of the over-specialized systems, where operations are easy, but little of interest is possible."* These systems are fitted very closely to specific tasks and will be difficult to use for anything outside the narrow scope for which they were designed. Modifying these systems to do things differently than the way provided leads to frustration and abandonment.

Domain-oriented design environments (*DODEs*) [Fischer, 1994] are a family of software programs that try to find the right mix between generality and specificity. For DODEs, we have developed a conceptual framework, an architecture, a process model, assessment schemes, and prototype systems for a variety of domains.

Seeding, Evolutionary Growth and Reseeding: A Process Model for Evolutionary Design

We live in a world characterized by evolution – that is, by ongoing processes of development, formation, or growth in both natural and human-created systems. Biology tells us that complex, natural systems are not created all at once but must evolve over time [Dawkins, 1987]. Evolutionary processes are ubiquitous and critical for technological innovations; this is particularly true for complex software systems that do not exist in a technological context alone but instead are embedded within dynamic human organizations.

An important reality of real software systems is that up to 75% percent of a system's cost over its lifetime is spent after the original system design is finished [CSTB, 1990]. Sustaining the usefulness of software systems differs from the traditional concept of "maintenance" because beyond repairing defects and fixing bugs, most of the efforts (an estimated 75% of the overall maintenance effort) are enhancement activities. The needs for enhancements are experienced by the skilled domain workers using these systems rather than by the system designers. One important claim behind our research is that skilled domain workers (at least the power users among them) should be empowered to create the required enhancements.

The *seeding, evolutionary growth, and reseeding (SER) model* [Fischer, 1998] is a descriptive and prescriptive model for collaborative design. It postulates that systems that evolve over a sustained time span must continually alternate between activities of unplanned evolution and periods of deliberate (re)structuring and enhancement. The SER model is based on the

observation that design problems in the real world require *open systems* that users can modify and evolve, and the changing requirements and functionality are determined through an iterative process of collaboration among multiple stakeholders. The fact that requirements cannot be completely specified before system development occurs has led to the following high-level guidelines underlying the SER model:

- *Software systems must evolve; they cannot be completely designed prior to use*. Design is a process that intertwines problem solving and problem framing [Schön, 1983]. Software users and designers will not be able to fully determine a system's desired functionality until that system is put to use.
- *Software systems must evolve at the hands of the users.* Users (not developers) experience a system's deficiencies; therefore, they have to play an important role in driving its evolution. Software systems need to contain mechanisms that allow users to modify their functionality and content [Burnett et al., 2004; Myers et al., 2006].
- *Software systems must be designed for evolution.* Despite the fact that evolution is no panacea and creates its own problems, there are strong reasons to increase the efforts and the costs to include mechanisms for evolution (such as end-user modifiability, tailorability, adaptability, design rationale, and making software "soft" [Lieberman et al., 2006]) in the original design of complex systems. Experience has shown that the costs saved in the initial development of a system by ignoring evolution will be spent several times over during the use of a system.

Software Design in Participation Cultures

Design for evolution provides foundations for such recent developments in software engineering as open source developments and meta-design extending the domain modeling approach to a collaborative domain construction approach. Design for evolution requires *"underdesign for emergent behavior"*: it focuses not on creating final solutions, but on creating spaces in which users as developers and designers can create their own solutions to fit their needs. Our work over the last few years has shifted and embraces the emerging Web 2.0 environments [O'Reilly, 2006] that provide the basis for a shift from an *industrialized information economy* (specialized in producing finished goods to be consumed passively) to a *networked information economy (in which all people are provided with the means to participate actively in personally meaningful problems)* [Benkler, 2006; Tapscott & Williams, 2006].

The *participation cultures* facilitated and supported the networked information economy are instrumental in the rapid emergence of new software systems that are based on the contributions by a community of users [von Hippel, 2005]. Systems such as Wikipedia , Flickr and Youtube [Tapscott & Williams, 2006], 3D Warehouse (http://sketchup.google.com/3dwarehouse/) , Scratch (http://scratch.mit.edu/), and open source software projects [Raymond & Young, 2001] are created through the collaboration of many contributors acting as equal partners by bringing their unique set of skills and expertise to shape their functionality and utility. In these *participative software systems* design does not end at the time of deployment and success hinges on continued participations and contributions of users at use time. Participative software systems need to be evolved continuously at the hand of users to achieve the best fit between the system and its ever-changing context of use, problems, domains, users, and communities of users.

In such systems, the roles of users and developers are blurred and design extends into use time. The design of participative software systems, therefore, presents a challenge of creating new methodological frameworks that re-define the roles of developers and users, re-distribute the design activities over the life cycle of the software systems, and give equal importance to the design of technical functionality and the design of social conditions for wide and sustained participation of users.

Meta-Design

Meta-design [Fischer & Giaccardi, 2006] is a design methodology to address the above challenges. Meta-design characterizes objectives, techniques, and processes for creating new media and environments that allow "owners of problems" to act as *designers*. A fundamental objective of meta-design is to create socio-technical environments [Mumford, 1987] that empower users to engage actively in the continuous development of systems rather than being restricted to the use of existing systems. Meta-design aims at defining and creating not only *technical infrastructures* for

the software system but also *social infrastructures* in which users can participate actively as co-designers to shape and reshape the socio-technical systems through collaboration.

In all design processes, two basic stages can be differentiated: design time and use time [Henderson & Kyng, 1991]. At *design time*, system developers (with or without user participation) create environments and tools for the *world as imagined* by them to anticipate users' needs and objectives. At *use time*, users use the system in the *world as experienced*. The bridging of these two stages into a unique *"design-in-use" continuum* creates an ongoing conversation both with the design material and among participants, which differentiates meta-design from other design methodologies such as user-centered design and participatory design. Meta-design provides a conceptual framework for *end-user development* [Lieberman et al., 2006; Myers et al., 2006] and *end-user software engineering* [Burnett et al., 2004].

Lessons Learned from Open Source Software for Participation Cultures

A systematic analysis of open source projects based on a meta-design perspective [Ye & Fischer, 2007] has allowed us to develop an initial framework for designing and supporting participation cultures.

Embracing Users as Co-Designers. To embrace users as co-designers, designers of *participative software systems (PSS)* need to pay attention to the fact that they are providing not only a solution to users, but also a solution space within which users can develop new solutions to their specific needs. The solution space contains components that owners of problems can use for their design activities, and determines the degree that they can evolve the original design. Currently available technology in software systems provides a variety of choices, ranging from the modification of options, the customization of menus and functions, the plug-in structure for extension, the published services for being mashed up with other services, the publication of system API for integration with other systems, and the source code that offers the highest freedom for user development. Meta-designers of PSS have to make a conscientious decision according to how much they want to get users involved.

Providing a Common Platform. Design contributions made by one individual user are limited because one particular user is interested only in creating solutions for his or her own needs. The power of distributed user design comes from the fact that the evolution of systems is pushed by a large number of users with diversified needs and skills who each make small contributions. For this to happen, users need to have a common platform supporting sharing and the integration of design solutions by others. Meta-designers need to either create an associated common toolkit or utilize a set of common tools widely available to all users to facilitate easy sharing and integration. The concept of open source software becomes possible only when software development tools (such as Emacs, Eclipse, and CVS) are widely available and are being used as standard tools by most software developers.

Enabling Legitimate Peripheral Participation. A transparent policy and procedure is needed to incorporate user contributions into PSSs. Users who made contributions need to see that their contributions make a recognizable influence on the system. Newcomers to a community must be able to engage in legitimate peripheral participation [Wenger, 1998]. To attract more users to become developers, the system architecture must be designed in a modularized way to create many relatively independent tasks with progressive difficulty so that newcomers can start to participate peripherally and move on gradually to take charge of more difficult tasks. The way a system is partitioned has consequences for both the efficiency of parallel development (a prerequisite for open source software) and the possibility of peripheral participation. The success of Linux is due in large part to its well-designed modularity. Another approach to afford peripheral participation is to *intentionally release under-designed systems* to users by leaving some non-critical parts unimplemented to facilitate easy participation (e.g.: the TODO lists of most open source systems create guidance for participation).

Sharing Control. Control needs to be shared between the original meta-designers of a PSS and the participating users. The roles that users can play are different, depending on their levels of involvement. Each level has its own responsibility and authority. Responsibility without authority cannot sustain users' interest in further involvement. When users change their roles in the PSS by making constant contributions, they should be granted the matching authority in the decision-making process that shapes the system. Meta-designers need to find a strategic way to transfer some of the control to users. Granting users controlling authority has two positive

impacts on sustaining user participation: (1) users who gain controlling authority become stakeholders, acquire ownership in the system, and are likely to make further contributions; and (2) having some authority will attract and encourage new users who want to influence the system development to make contributions. Successful open source software projects invariably select skilful "user-turned-developers" and grant them access privileges to contribute directly to the source base.

Promoting Mutual Learning and Support. Users have different levels of skill and knowledge about the system. To get involved in contributing to the system or using the system, they need to learn many things. Peer users are important learning resources. A PSS should be accompanied with knowledge sharing mechanisms that encourage users to learn from each other. In open source software projects, mailing lists, discussion forums, and chat rooms provide an important platform for knowledge transfer and exchange among peer users.

Fostering a Social Rewarding and Recognition Structure. Motivation [Fischer et al., 2004] is essential for the success of PSSs. Factors that affect motivation are both intrinsic and extrinsic. The precondition for motivating users to get involved in contribution is that they must derive an intrinsic satisfaction in their involvement by shaping the software system to solve their problems. Intrinsic motivation is positively reinforced and amplified when social structure and conventions of the community recognize and reward the contributions of its members.

The social fabric inherent in open source communities reinforces the intrinsic motivation for participating. Members close to the center of the community enjoy better visibility and reputations than do peripheral members. As new members contribute to the system and the community, they are rewarded with higher recognition, trust, and influence in the community. Rewarding contributing members with higher recognition and more important roles is also important for the sustainability of the community as well as system development because it is the way that the community reproduces itself. Developers of PSSs need to establish a social norm in the user communities by recognizing publicly contributing users and promoting their social status in the community by granting matching authority.

Distribution and Diversity in Communities

In extended and distributed software design projects, stakeholders from many different domains must coordinate their efforts despite large separations of time and distance. In such projects, collaboration is crucial for success, yet it is difficult to achieve. Complexity arises from the need to synthesize different perspectives, exploit conceptual collisions between concepts and ideas coming from different disciplines, manage large amounts of information, and understand the design decisions that have determined the long-term evolution of a designed artifact.

Cultures of participation thrive on the diversity of perspectives included by making all voices heard. They require constructive dialogs between individuals negotiating their differences while creating their shared voice and vision. Exploring spatial, temporal, and conceptual distances [Fischer, 2005] will provide additional foundations for rethinking software design in participation cultures.

Voices from Different Places: Spatial Distance. Bringing spatially distributed people together with the support of computer-mediated communication allows *shared concerns* rather than shared location to be the defining feature of a group of people interacting with each other. On the one hand, communication technologies enable new forms of collaborative work and they exploit local knowledge in a globalized world. On the other hand, closely coupled work can still be difficult to support at a distance, and critical stages of collaborative work, such as establishing mutual trust, require some level of face-to-face interaction [Olson & Olson, 2001].

Voices from the Past: Temporal Distance. Design processes often take place over many years, with initial design followed by extended periods of evolution and redesign. Design artifacts (including systems that support design tasks, such as reuse environments) are not designed once and for all, but instead evolve over long periods of time [Dawkins, 1987]. For example, when a new device or technology emerges, most computer networks are enhanced and updated rather than redesigned completely from scratch. Much of the work in ongoing design projects is done as redesign and evolution; often, the people doing this work were not members of the original design team. To be able to do this work well, or sometimes at all, requires that these people "collaborate" with the original designers of the artifact.

We have developed *reuse-conducive development environments* [Ye & Fischer, 2005], which encourage and enable software developers to reuse through the smooth integration of reuse repository systems and development environments. *CodeBroker (*a reuse-conducive development environment) autonomously locates and delivers task-relevant and personalized components into the current software development environment. Empirical evaluations of *CodeBroker* have shown that the system is effective in promoting reuse by enabling software developers to reuse components unknown to them, reducing the difficulties in locating components, and augmenting the programming capability of software developers.

Voices from Different Communities: Conceptual Distances. Cultures of participation are social structures that enable groups of people to share knowledge and resources in support of collaborative design. Different communities (such as communities of practice and communities of interest) grow around different types of design practices. *Communities of Practice (CoPs)* [Wenger, 1998] consist of practitioners who work as a community in a certain domain undertaking similar work. Examples of CoPs are architects, urban planners, research groups, software developers, and end-users. CoPs gain their strength from shared knowledge and experience. However, they face the danger of *group-think* [Janis, 1972]: the boundaries of domain-specific ontologies and tools that are empowering to insiders are often barriers for outsiders and newcomers.

Communities of Interest (CoIs) [Fischer, 2001] bring together stakeholders from different CoPs to solve specific design problems of common concern. They can be thought of as "communities-of-communities" [Brown & Duguid, 2000]. Examples of CoIs are (1) a group of citizens and experts interested in urban planning, (2) representatives of the creative practices and new media designers to create new shared visions and artifacts between art and technology [National-Research-Council, 2003]; and (3) software designers, caregivers, and psychologists to create new socio-technical environments for people with cognitive disabilities. Fundamental challenges facing CoIs are found in building a shared understanding of the task-at-hand, which often does not exist at the beginning but is evolved incrementally and collaboratively and emerges in people's minds and in external artifacts. Members of CoIs must learn to communicate with and learn from others who have different perspectives and perhaps different vocabularies to describe their ideas and to establish a common ground.

Automate and (or versus?) Informate

The essence of socio-technical systems supporting cultures of participation is to facilitate a successful combination of human skills and computing power that allows humans and computers to do design in a manner that cannot be done by either designers or computers alone. To achieve this objective requires determining the right mix between *"automate"* and *"informate"* [Zuboff, 1988] needed for specific situations [Billings, 1991]). In developing new classes of socio-technical systems (including notations, tools, and collaboration support), the following questions need to be asked:

- Which parts of tasks and responsibilities have to be exercised by *human beings* because they are better reserved for a skilled or experienced human mind?
- Which ones should be taken over by or aided by *automated computational components*?
- Which kind of collaboration environments are required so that all involved stakeholders can interact effectively with each other and the automated computational components?

Although some processes, such as compiling a program, can be and should be fully automated, the degree of automation for other design activities is less clear. Reflecting on our work over the last couple of decades and paying attention to the questions asked above, we are convinced that the most promising frameworks can be built on an *"and"* rather than on a *"versus"* relationship between automate and informate. In many situations, informating can build on automation. Domain-oriented design environments put owners of problems in charge but achieve this objective by incorporating many automated components.

The opportunities to exploit automation for the informating capacity of socio-technical environments in participation cultures go even further and have only partially been explored. The traces left by the communication and collaboration activities can be automatically extracted, analyzed, and visualized for task modeling , increase in awareness, personalization of information, recommendations, and suggestions. These components and processes represent powerful new technologies, which can be equally misused for drowning people in irrelevant and

unwanted information and privacy violations. The research agenda to explore and understand these critical activities and trade-offs is full of interesting problems, and the *Journal of Automated Software Engineering* hopefully will continue to attract and document progress in these areas.

Acknowledgements

The author thanks the members of the Center for LifeLong Learning & Design (L3D) at the University of Colorado, who have made major contributions to the research described in this paper. Specifically valuable and important contributions have been made by the numerous PhD students who developed innovative ideas, frameworks, and prototype systems to explore new territory in software engineering and who have continued their research activities after graduation. I am grateful to the editors and reviewers of the Journal "Automated Software Engineering" who have found our work relevant enough to be published in several articles in their journal.
Our research in human-centered software design was supported by a substantial number of research grants from the National Science Foundation and by SRA, Inc., Tokyo, Japan.

References

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977) *A Pattern Language: Towns, Buildings, Construction,* Oxford University Press, New York.

Belady, L., & Lehman, M. (1985) *Program Evolution Processes of Software Change*, Academic Press, London, UK.

Benkler, Y. (2006) *The Wealth of Networks: How Social Production Transforms Markets and Freedom,* Yale University Press, New Haven, CT.

Billings, C. E. (1991) *Human-Centered Aircraft Automation: A Concept and Guidelines,* NASA Technical Memorandum, Report No. 103885, NASA Ames Research Center.

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

Burnett, M., Cook, C., & Rothermel, G. (2004) "End-User Software Engineering,," *Communications of the ACM,* 47(9), pp. 53-58.

CSTB (1990) "Scaling Up: A Research Agenda for Software Engineering (Computer Science and Technology Board)," *Communications of the ACM,* 33(3), pp. 281-293.

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

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

Fischer, G. (1998) "Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments," *Automated Software Engineering,* 5(4), pp. 447-464.

Fischer, G. (2001) "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. (2003) "Desert Island: Software Engineering — A Human Activity," *International Journal Automated Software Engineering,* Kluwer Academic Publishers, Dordrecht, The Netherlands, 10(2), pp. 233-237.

Fischer, G. (2005) "Distances and Diversity: Sources for Social Creativity," *Proceedings of Creativity & Cognition*, London, April, pp. 128-136.

Fischer, G., & Giaccardi, E. (2006) "Meta-Design: A Framework for the Future of End User Development." In H. Lieberman, F. Paternò, & V. Wulf (Eds.), *End User Development: Empowering People to Flexibly Employ Advanced Information and Communication Technology,* Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 427-457.

Fischer, G., & Lemke, A. C. (1988) "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction,* 3(3), pp. 179-222.

Fischer, G., Scharff, E., & Ye, Y. (2004) "Fostering Social Creativity by Increasing Social Capital." In M. Huysman, & V. Wulf (Eds.), *Social Capital and Information Technology,* MIT Press, Cambridge, MA, pp. 355-399.

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

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.

Illich, I. (1973) *Tools for Conviviality,* Harper and Row, New York.

Janis, I. (1972) *Victims of Groupthink,* Houghton Mifflin, Boston.

Lee, L. (1992) *The Day the Phones Stopped,* Donald I. Fine, Inc., New York.

Levy, F., & Murnane, R. J. (2004) *The New Division of Labor: How Computers Are Creating the Next Job Market,* Princeton University Press, Princeton, NJ.

Lieberman, H., Paterno, F., & Wulf, V. (Eds.) (2006) *End User Development – Empowering People to Flexibly Employ Advanced Information and Communication Technology,* Kluwer Publishers, Dordrecht, The Netherlands.

Mumford, E. (1987) "Sociotechnical Systems Design: Evolving Theory and Practice." In G. Bjerknes, P. Ehn, & M. Kyng (Eds.), *Computers and Democracy,* Avebury, Aldershot, UK, pp. 59-76.

Myers, B. A., Ko, A. J., & Burnett, M. M. (2006) "Invited Research Overview: End-User Programming." In *Human Factors in Computing Systems, CHI'2006 (Montreal)*, pp. 75-80.

National-Research-Council (2003) *Beyond Productivity: Information Technology, Innovation, and Creativity,* National Academy Press, Washington, DC.

O'Reilly, T. (2006) *What Is Web 2.0 - Design Patterns and Business Models for the Next Generation of Software,* Available at http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

Olson, G. M., & Olson, J. S. (2001) "Distance Matters." In J. M. Carroll (Ed.), *Human-Computer Interaction in the New Millennium,* ACM Press, New York, pp. 397-417.

Perlis, A. J. (1982) "Epigrams on Programming." In *SIGPLAN Notices*, pp. 7-13.

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.

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

Shaw, M. (1989) "Maybe Your Next Programming Language Shouldn't Be a Programming Language." In Computer Science Technology Board (Ed.), *Scaling Up: A Research Agenda for Software Engineering,* National Academy Press, Washington, DC, pp. 75-82.

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

Tapscott, D., & Williams, A. D. (2006) *Wikinomics: How Mass Collaboration Changes Everything,* Portofolio, Penguin Group, New York.

von Hippel, E. (2005) *Democratizing Innovation,* MIT Press, Cambridge, MA.

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

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

Ye, Y., & Fischer, G. (2005) "Reuse-Conducive Development Environments," *International Journal Automated Software Engineering,* Kluwer Academic Publishers, Dordrecht, The Netherlands, 12(2), pp. 199-235.

Ye, Y., & Fischer, G. (2007) "Designing for Participation in Socio-Technical Software Systems." In C. Stephanidis (Ed.), *Proceedings of 4th International Conference on Universal Access in Human-Computer Interaction (Beijing, China),* Springer, Heidelberg, pp. 312-321.

Zuboff, S. (1988) *In the Age of the Smart Machine,* Basic Books, Inc., New York.