Metadesign:
Guidelines for Supporting Domain Experts in Software Development

Gerhard Fischer, *University of Colorado, Boulder*

Kumiyo Nakakoji, *SRA Key Technology Laboratory*

Yunwen Ye, *Software Research Associates*

> The metadesign framework encompasses objectives, techniques, and processes for creating computational tools that let domain experts act as designers.

In the past few decades, the primary goal of most software systems has been to improve productivity in various domains by supporting established processes and actions. Now, software systems also have become important instruments for creating new practices in an infinite number of application domains by letting domain experts creatively explore, frame, and solve problems. So, more and more people are not only using software but also getting involved in developing it to meet their ever-changing needs.

Our collaborative research activities in software development (at the University of Colorado's Center for Lifelong Learning and Design, the University of Tokyo, and Software Research Associates) have focused on understanding the implications of the quickly disappearing distinction between users and developers. We've also concentrated on establishing new software development methodologies by viewing software systems as continuously evolving sociotechnical systems[1] driven by design activities of both professional software engineers and users.
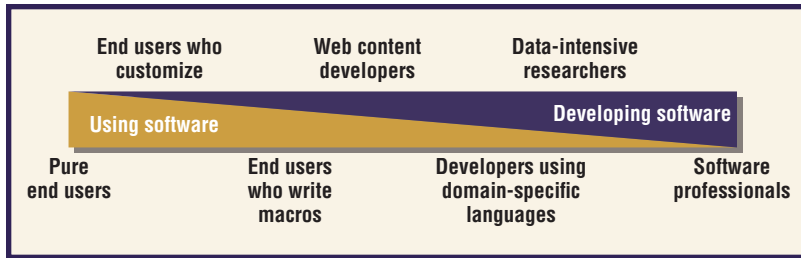
We believe that domain experts, as the owners of problems, need to be in charge of developing the software they require. Toward that end, we've created the *metadesign* framework, which reformulates software development activities as a continuum of different degrees of design and use. In addition, on the basis of our research and our findings in the research literature, we've developed a set of guidelines for supporting domain experts in software development.

Putting the Problem Owners in Charge

As early as 1988, Bill Curtis and his colleagues[2] had identified fundamental challenges for software engineering and a rationale for why an understanding of and support for end-user software engineering[3] are a necessity rather than a luxury. They identified the following three challenges, which motivated our development of the metadesign framework.

The first challenge was the growing importance of application domain knowledge for most software systems and that this knowledge is held by domain experts rather than by software developers, who suffer from a "thin spread of application domain knowledge."[2] This finding provided the foundation for our research in *domain-oriented design environments* (DODEs),[4] which we discuss later.

The second challenge was the need for open, evolvable systems that can adjust to fluctuating, conflicting requirements. Such requirements will lead over time to mismatches between an evolving

End users who customize	Web content developers	Data-intensive researchers	
Using software		Developing software	
Pure end users	End users who write macros	Developers using domain-specific languages	Software professionals

Figure 1. The spectrum of software-related activities. The once sharp distinction between users and developers of software is fading away, and many users are starting to take control of shaping software to their own needs through their own development activities.

world and the software system modeling this world. This finding provided the foundation for Gerhard Fischer and his colleagues' *seeding, evolutionary growth, reseeding* (SER) model,[5] which we also discuss later.

The final challenge was the need to support communication and coordination in a richer ecology of participants with different interests, skills, and background knowledge. Curtis and his colleagues realized that system development is difficult not because of the technical problems' complexity but because of the need for mutual understanding and common ground between all participating stakeholders. This finding motivated our emphasis on the coevolution of systems, communities, and individuals.[6]

Our recent interview with a geoscientist highlights these challenges' importance. He uses several domain-specific software systems to analyze his research data. However, those systems cannot provide complete solutions to his problems as his research unfolds and his understanding of the problem, data, and results progresses. He said,

> I spend on average an hour every day developing software for myself to analyze the data I collected because there is not any available software. Even if there is a software developer sitting next to me, it would not be of much help because my needs vary as my research progresses and I cannot clearly explain what I want to do at any moment. Even if the software developer can manage to write a program for me, I will not know if he or she has done it right without looking at the code.

He continued,

> So I spent three months to gain enough programming knowledge to get by. Software development has now become an essential task of my research, but I do not consider myself a software developer, and I don't know many other things about software development.

Clearly, this geoscientist isn't a professional soft-

ware engineer, and he doesn't intend to become one. He isn't a mere end user, either, because he engages regularly in intensive software development that goes beyond what most end-user programming environments support. He has acquired excellent programming skills for solving his own problems with the tools that professional software engineers use (for example, Unix, shell, and vi).

Software development is no longer the exclusive activity of professional software engineers. Many domain experts such as this geoscientist are engaging in software development as intensive and technically challenging as that performed by many professional software engineers. However, they're creating software for their own rather than others' problems, and as an instrument for a larger context rather than as just an end artifact to be delivered.

Figure 1 describes a richer ecology of people's roles in developing and using software. Professional software engineers and pure end users define the endpoints of a continuum of computer users. The former like computers because they can program, and the latter because they get their work done with computers. Most users generally won't build tools of the quality a professional software engineer would. However, if a tool doesn't satisfy the needs or tastes of users, who know best what their requirements are, they should be able to develop their own solutions.

The World Wide Web's short history provides convincing evidence of the quick dissolution of the sharp distinction of design and use. In the Web's first decade, a clear separation between developers and users was predominant, in which users used whatever professional Web developers gave them. As we enter the Web 2.0 era, we're witnessing a much richer ecology of user participation. Users now also participate in various development activities that continuously evolve Web applications through adoption, adaptation, appropriation, and mashing-up of existing Web systems.

On the basis of this reframing of software development, it follows that the dichotomy of design and use doesn't hold for many software systems. Software systems should support a continuum of design-in-use activities by allowing, enabling, and leveraging users' development activities while they're experiencing the world through software systems.

The Metadesign Framework
Drawing from our research experience, we've developed metadesign as a framework to explore objectives, techniques, and processes for creating

computational tools that let domain experts who are the "owners of problems" act as designers.[7] A fundamental objective of metadesign is to create sociotechnical environments that empower domain experts to engage actively in the continuous development of systems, rather than restrict them to using existing systems. Metadesign defines and creates not only technical infrastructures for the software system but also social infrastructures in which users can participate actively as codesigners to shape and reshape the sociotechnical systems through collaboration.[8]

All human-made artifacts (including software systems) undergo two basic stages: design time and use time. At design time, designers or producers create artifacts for the world as they imagine it, to anticipate users' needs and objectives. At use time, users employ the artifacts to accomplish their tasks in the world as they experience it.

Many design and engineering theories assume that you can imagine, capture, and specify the world as experienced at design time and that all major design activities therefore end when you deliver the artifacts for use. The breakdown between the world as experienced and the world as imagined is more pronounced in software, especially for software that supports domain experts engaged in creative knowledge work. The real-world problems that domain experts face are often ill defined and wicked.[9] A wicked problem has complex interdependencies of many contradictory and changing aspects. Attempts to solve one aspect of the problem often create new problems; therefore, a wicked problem can't be easily understood before solutions are attempted. Software systems modeling these problems are never static, for two reasons. First, the world changes and new requirements emerge. Second, domain experts change their work practices over time, and their understanding and use of a system will be very different after a month and certainly after several years.

The SER Process Model

Our research interest is in designing the social and technical infrastructures in which new forms of collaborative design can take place. For most design domains we've studied over many years, the knowledge to understand, frame, and solve problems isn't given but is constructed and evolves during problem framing and problem solving.[5]

Seeding. In the past, large, complex information systems were built as "complete" artifacts through the large efforts of a few people. Instead of attempting to build complete systems, the SER model advocates building seeds that evolve over time through the small contributions of many people.

A seed is based on an initial understanding and framing of the problem. Environment developers create it to be as complete as possible for future users. However, the understanding of the problem can't be truly complete because domain experts' knowledge work is situated and tacit. Furthermore, the constant changes in the environment in which the system is embedded will breed new needs, and the introduction of a computational system itself generates changes in professional practices and sociotechnical environments. So, the initial seed must be continuously adapted to the new understanding and environments.

Evolutionary growth. This phase is one of decentralized evolution as domain experts use and extend the seed to do their work or explore new problems. This phase doesn't directly involve professional environment builders because the focus has shifted to domain experts' problem-framing and problem-solving activities. Instead, those domain experts who have a direct stake in the problem at hand perform the development.

During this phase, the seed plays two roles simultaneously. First, it provides work resources (solutions accumulated from prior use). Second, it accumulates work products as each project contributes a new solution and extension to the seed. In this phase, users focus on creating problem-specific solutions rather than general solutions. So, these solutions might not be well integrated with the rest of the solution in the seed.

Reseeding. This phase involves a deliberate, centralized effort to organize, formalize, and generalize solutions and artifacts created during evolutionary growth. It aims to create an information space in which useful solutions can be easily found, reused, and extended. As in the seeding phase, professional software developers are needed to perform substantial system and solution-space modifications. However, users must also participate because only they can judge what solutions are useful and what structures will serve their work practices.

Coevolution of System, Community, and Individuals

During evolutionary growth, the software system doesn't evolve independently of the surrounding social infrastructure's change and growth. It's better to view the software system's developers and users as a knowledge community formed by their various attachments with the system.

> Domain experts like to interact with problems, not computer systems.

A typical example is the open source software (OSS) resulting from the removal of the sharp distinction between design and use of software systems. Our systematic analysis of several OSS systems and communities revealed that people take on a variety of roles: passive users, readers, bug reporters, bug fixers, peripheral developers, active developers, and project leaders.[10]

We further observed that an OSS system's evolution is driven by interaction between community members and the system and among members with different roles. As community members interact with the system, their knowledge of the system increases and their work practices change. Some members start taking different roles in system development by reporting use experience, contents, bugs, extensions, adaptations, or code patches. Such contributions further evolve the system.

System use also involves individuals interacting with the community by sharing experience and knowledge regarding the system. Such interaction changes the individual's relations with other members in terms of personal relationships and social recognition. It also changes the community's social fabric as some members migrate gradually toward the community's center when their contributions and knowledge are recognized. The community's existence provides users and developers with the sense and experience of common purpose and mutual support in evolving the system. In many situations, it replaces common background or geographic proximity with a sense of well-defined purpose, shared concerns, and the successful common pursuit of these.[6]

Metadesign Guidelines for Sociotechnical Systems

Here we present our guidelines for using the metadesign framework to design sociotechnical systems.

Support Human-Problem Interaction

Because domain experts aren't interested in computers per se, they aren't inclined to spend considerable effort to learn general software development skills but instead prefer to focus on their domain problems. In other words, they like to interact with problems, not computer systems.

Increasing domain specificity is an effective way to create computer systems that support human-problem interaction. We've designed and evaluated a series of DODEs in various domains.[4] These environments help domain experts understand their design problems better while exploring design solutions, instead of translating existing design solutions into computational representa-

tions. DODEs provide a rich combination of system components:

- a specification component that lets domain experts specify their problems using their own concepts,
- a construction kit consisting of basic solution elements,
- an argumentation and critiquing mechanism that stimulates users to reflect on their design decisions,
- a simulation component that helps visualize the effects of design decisions, and
- a catalog of design solutions to the problems that designers know when doing the design.

KID (Knowing-in-Design) is a DODE for kitchen design. Through its specification component, KID first asks a kitchen designer several questions about his or her cooking habits and lifestyle. On the basis of this information, KID presents relevant design examples from its catalog for the designer to use as a starting point. The designer can then use the construction kit to modify the design. If certain design decisions violate built-in design rules or regulations, the critiquing mechanism points out and explains the potential problem. However, the designer can overwrite such critiques with his or her design rationale, which is added to the KID knowledge base.[4]

Underdesign for Emergent Behavior

Metadesign focuses not on creating final solutions but on creating solution spaces in which users can create their own solutions to fit their needs. Systems should be underdesigned so that users don't treat them as a finished product but view them as continuous beta versions that are open to facilitate and incorporate emergent design behaviors during use.

Underdesign doesn't mean that the seed's creator transfers his or her design responsibilities to the users and forces them into a do-it-yourself situation. Instead, it requires creating tools that users can employ to solve those well-defined problems and supporting "hackability" and "remixability" by providing metatools that users can employ for occasions not envisioned at design time. So, underdesign isn't less design but more design.

Our experiments with underdesign in several projects have tried to manage the tension between constraint and freedom and between rigor and relevance. For example, KID also embraces this principle. Each solution in the design catalog solves a typical design problem and can be used as is. However, users can also decompose it into smaller design

elements that they can combine with other design elements, using the construction kit.

Web browsers' view-source functionality is another excellent example of supporting emergent behaviors. It enables an interested user to examine, learn, and extract a partial design of a well-designed Web page for his or her own use. This principle is further extended in Mozilla's Firebug system, which lets users inspect dynamic Web pages' inner source (Cascading Style Sheet styles, Document Object Model structures, and JavaScript code). The view-source functionality and Firebug system make a Web page not only a solution to a problem but also a design space to be further explored, extended, and mixed.

Enable Legitimate Peripheral Participation

Transparent policies and procedures are needed for incorporating user contributions into the software systems. Users who make contributions need to see that their contributions have a recognizable influence on the system.

In particular, newcomers to a community must be able to engage in legitimate peripheral participation.[11] To attract more users to become developers, the system architecture must be modularized to create many relatively independent tasks with progressive difficulty so that newcomers can start participating peripherally and move on gradually to take charge of more difficult tasks. How 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. Linux's success is due largely to its well-designed modularity. Studies have shown that most current developers of the Linux kernel started by working on its relatively independent device-driver modules.[12]

Another way to afford peripheral participation is to intentionally release unfinished systems to users by leaving some noncritical parts unimplemented to facilitate easy participation. For example, the to-do lists of most OSSs create guidance for participation. A third way is to foster satellite communities for incubation subprojects, which will be incorporated into the main project when they mature.[10]

Share Control

A software system's original metadesigners must share control with the participating users. Users can play different roles, depending on their levels of involvement. Each level has its own responsibility and authority. Responsibility without authority won't sustain users' interest in further involvement. When users change their roles in the community by making constant contributions, they should be granted the matching authority in the decision-making process that shapes the system.

Metadesigners need to find a strategic way to transfer some control to users. Granting users controlling authority helps sustain user participation and system evolution in two ways:

- Those users become stakeholders, acquire ownership in the system, and will likely make further contributions.
- Granting authority attracts users who want to influence system development and encourages them to contribute.

Successful OSS projects invariably select skillful "users-turned-developers" and grant them access privileges to contribute directly to the source base.

Our analysis of OSS systems and communities revealed that different policies of sharing control with other users have resulted in different evolution patterns for those systems and communities.[10] For example, tighter control of granting write accessibility to community users resulted in systems' slower evolution.

Promote 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 it, they need to learn many things. Peer users are important learning resources. A metadesigned sociotechnical system should have associated knowledge-sharing mechanisms that encourage users to learn from each other. In OSS projects, mailing lists, discussion forums, and chat rooms provide an important platform for knowledge transfer and exchange among peer users.

We've tried to extend the traditional online support system with a built-in peer learning platform in the STeP_IN (*Sociotechnical Platform for In Situ Networking*) project.[13] STeP_IN enhances the traditional Java documentation system with an "ask expert" button through which a user can post a question to other users who have demonstrated their expertise on the Java API library component.

Reward and Recognize Contributions

Motivation is essential for the success of user participation in the evolution of metadesigned systems. Human beings are diversely motivated. We act not only for material gain but also for psychological well-being, social integration and connectedness, social capital, recognition, and improving our standing in a reputation economy. Horst Rittel
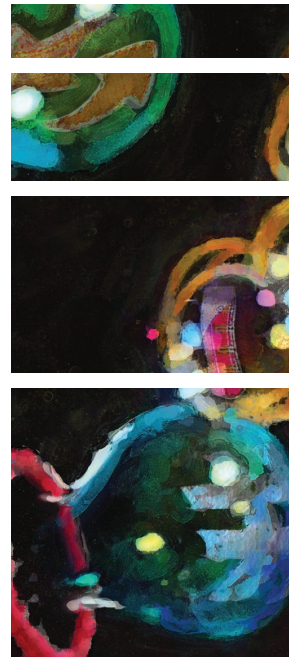
articulated the motivation for going the extra step to engage in evolving software systems:

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.[9]

Both intrinsic and extrinsic factors affect motivation. The precondition for motivating users to contribute 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 the community's social structure and conventions recognize and reward its members' contributions.

In STeP_IN, we've developed a set of methods for computing social obligations and expectations. These methods are based on social capital as a mechanism for motivating users' participation in the continual evolution of the online collaborative learning platform.

Foster Reflective Communities

Complex design problems require more knowledge than any one person possesses because the knowledge relevant to a problem is usually distributed among many domain experts. Creating a shared understanding among domain experts requires bringing together different, often controversial viewpoints and can lead to new insights, new ideas, and new artifacts. Domain experts need to rely on other people's knowledge and on external information. The relevant knowledge for complex design activities is distributed among multiple human beings and artifacts, bringing together different knowledge sources, none of which has the final authority. By exploiting the "symmetry of ignorance"[9] and mutual competency, stakeholders can learn from each other.

Early studies already identified that end-user development is more successful when it's supported by collaborative work practices rather than when it focuses on individuals.[14] The studies observed the emergence of "gardeners" and "local developers." Such people are technically interested and sophisticated enough to perform system modifications that a user community needs but that other end users can't or don't want to perform.

Possible Pitfalls

Metadesign's goal isn't to let untrained people develop and evolve sophisticated software systems but

to put the problems' owners in charge and make them independent of "high-tech scribes."[8] One critical challenge in creating software systems is to achieve the best fit between the system and its ever-changing context of use, problems, domains, users, and user communities. The metadesign framework sees problem owners as the ultimate source to achieve the fittest software solutions to their problems. It also seeks a systematic way to support them as active contributors and designers while they use the system.

This approach creates inherent tensions between standardization and improvisation. For example, *SAP Info* warns of the perils of customer modifications:

Every customer modification implies costs because it has to be maintained by the customer. Each time a support package is imported there is a risk that the customer modification may have to be adjusted or re-implemented. To reduce the costs of such on-going maintenance of customer-specific changes, one of the key targets during an upgrade should be to return to the SAP standard wherever this is possible.[15]

Finding the right balance between standardization (which can suppress innovation and creativity) and improvisation (which can lead to a Babel of different, incompatible versions) is a particular challenge in open source environments, in which forking has often led developers in different directions.

Insights for End-User Software Engineering

The basic assumptions behind end-user software engineering research are that end users are educated differently from professional software developers and face different motivations and work constraints.[16] They aren't likely interested in concepts such as quality control, formal development, and rigorous testing strategies, and aren't willing or interested to invest extensive time learning about these things.[17] Metadesign provides a framework to think about how end-user software engineering is fundamentally different from professional software engineering, on the basis of the following four observations.

Requirements Generation

End-user programming generates requirements differently than professional software development does. Because the developers are also the users, there's no need for elaborate requirements analysis

before they construct a software system. Rapid requirements changes needn't be avoided; on the contrary, they're desired because end-user programmers use the computer to explore the new possibilities and find the "undreamed-of requirements." Traditional software engineering has worked hard to control and manage requirement changes. However, for domain experts, the issue is the opposite: how to encourage this exploration and discovery of new requirements and possibilities.

Software Testing

End-user programmers also conduct software testing differently. Because the domain experts themselves are the primary users, complete testing isn't as important as when the developers aren't the users. In such settings, instead of developers creating test plans beforehand, test plans could be automatically generated through the capture of testing activities or in a managed testing environment.

Collaboration

In end-user programming, collaboration takes place along different dimensions. In metadesign environments, predefined project teams don't exist. Collaboration takes place only when people interested in similar software become aware of each other and have the means to collaborate, or when a system that one domain expert developed is picked up by others with a similar problem. Collaboration is much more spontaneous and opportunistic than planned. Collaboration is often more limited in time because domain experts might lose interest in the software once they solve their own problem.

Knowledge and Skill Acquisition

Finally, the path to the acquisition of software development knowledge and skill is different. Owing to the lack of interest in software per se and the lack of professional training, domain experts are more likely to acquire software knowledge in a piecemeal, demand-driven manner. Their knowledge is more fragmentary than systematic. New approaches to learning software development must be investigated and supported.

With computers and software becoming pervasive, many domain experts have started to develop or adapt sophisticated software systems as an integral part of their work to fully utilize the power of the computer. They aren't professionally educated as software engineers but spend a great deal of their time creating software systems for their own work. Given

how domain experts' needs, goals, and education differ from those of professional software engineers, end-user software engineering research shouldn't be based on a scaled-down version of, or a simple transfer from, current software engineering principles. Metadesign is a new conceptual framework for understanding such fundamental differences. It aims to provide guidelines for designing sociotechnical environments that support the efficient development of useful software systems by those domain experts, the population of which will likely be much larger than that of professional software engineers. ⌘

References

1. E. Mumford, "A Socio-technical Approach to Systems Design," *Requirements Eng.*, vol. 5, 2000, pp. 59–77.
2. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM*, vol. 31, no. 11, 1988, pp. 1268–1287.
3. M. Burnett et al., "End-User Software Engineering with Assertions in the Spreadsheet Paradigm," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 93–103.
4. G. Fischer, "Domain-Oriented Design Environments," *Automated Software Eng.*, vol. 1, no. 2, 1994, pp. 177–203.
5. G. Fischer et al., "Seeding, Evolutionary Growth and Reseeding: The Incremental Development of Collaborative Design Environments," *Coordination Theory and Collaboration Technology*, G.M. Olson, T.W. Malone, and J.B. Smith, eds., Lawrence Erlbaum Associates, 2001, pp. 447–472.
6. K. Nakakoji, Y. Yamamoto, and Y. Ye, "Supporting Software Development as Knowledge Community Evolution," *Proc. ACM CSCW Workshop Supporting the Social Side of Large-Scale Software Development*, ACM Press, 2006, pp. 31–34.
7. G. Fischer et al., "Meta-design: A Manifesto for End-User Development," *Comm. ACM*, vol. 47, no. 9, 2004, pp. 33–37.
8. G. Fischer and E. Giaccardi, "Meta-design: A Frame-

work for the Future of End User Development," *End User Development*, H. Lieberman, F. Paternò, and V. Wulf, eds., Kluwer Academic, 2006, pp. 427–457.

9. H. Rittel, "Second-Generation Design Methods," *Developments in Design Methodology*, N. Cross, ed., John Wiley & Sons, 1984, pp. 317–327.

10. Y. Ye and K. Kishida, "Toward an Understanding of the Motivation of Open Source Software Developers," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 419–429.

11. E. Wenger, *Communities of Practice: Learning, Meaning, and Identity*, Cambridge Univ. Press, 1998.

12. G. von Krogh, S. Spaeth, and K.R. Lakhani, "Community, Joining, and Specialization in Open Source Software Innovation: A Case Study," *Research Policy*, vol. 32, 2003, pp. 1217–1241.

13. Y. Ye, Y. Yamamoto, and K. Nakakoji, "A Socio-technical Framework for Supporting Programmers," *Proc. 2007 ACM Symp. Foundations of Software Eng.* (FSE 07), ACM Press, 2007, pp. 351–360.

14. B.A. Nardi, *A Small Matter of Programming*, MIT Press, 1993.

15. "Reduce the Number of Customer Modifications," *SAP Info*, July 2003, p. 33.

16. M. Burnett, "What Is End-User Software Engineering and Why Does It Matter?" *End-User Development*, V. Pipek et al., eds., Springer, 2009, pp. 15–28.

17. B.A. Myers, A.J. Ko, and M.M. Burnett, "Invited Research Overview: End-User Programming," *Proc. Conf. Human Factors in Computing Systems* (CHI 06), ACM Press, 2006, pp. 75–80.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

About the Authors

Gerhard Fischer is a professor of computer science, a fellow of the Institute of Cognitive Science, and the director of the Center for LifeLong Learning and Design at the University of Colorado at Boulder. His research focuses on learning, working, and collaborating with new media; human-computer interaction; the science of design (open systems, metadesign, and 2.0 environments); creativity and IT; and transdisciplinary collaboration and education. Fischer has a Habilitation in computer science from the University of Stuttgart. He's a member of the CHI (Computer-Human Interaction) Academy, ACM, and American Educational Research Association. Contact him at gerhard@colorado.edu; http://l3d.cs.colorado.edu/~gerhard.

Kumiyo Nakakoji is a full professor at the University of Tokyo's Research Center for Advanced Science and Technology and the director of the SRA Key Technology Laboratory. Her research interest is knowledge-interaction design, which is a framework for the design and development of computational tools nurturing creative knowledge work, such as scholarly work, software development, interaction design, online-community design, and experience design for consumer electronics. Nakakoji has a PhD in computer science from the University of Colorado at Boulder and is certified by the Institute of Cognitive Science. She's a member of the ACM, IEEE Computer Society, Information Processing Society of Japan, Japanese Society for Artificial Intelligence, Human Interface Society, Japanese Cognitive Science Society, and Japanese Society for the Science of Design. Contact her at kumiyo@kid.rcast.u-tokyo.ac.jp.

Yunwen Ye is a manager at Software Research Associates. His research interests include cognitive and social aspects of software development, developer-centered software development environments, and sociotechnical support for knowledge creation and collaboration. Ye has a PhD in computer science from the University of Colorado at Boulder. He's a member of the Japanese Society for Artificial Intelligence and the Software Engineers Association. Contact him at ye@sra.co.jp.