University of Colorado at Boulder

Center for LifeLong Learning and Design (L3D)

Department of Computer Science

ECOT 717 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-1592, FAX: (303) 492-2844

Domain-Oriented Design Environments Mini-Tutorial

Gerhard Fischer

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

Proceedings of the 18th International Conference on Software Engineering, Computer Society, Berlin, Germany, March 1996, pp. 517-520

"Domain-Oriented Design Environments"

Gerhard Fischer
Center for LifeLong Learning and Design (L3D)
Department of Computer
Science and Institute of Cognitive Science
University of Colorado at Boulder

Extended Abstract

Domain-Oriented Design Environments (DODEs) [Fischer 1994] emphasize a human-centered and domain-oriented approach facilitating communication about evolving systems among all stakeholders. The domain orientation 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.

Design [Simon 1981] in the context of our research approach refers to the broad endeavor of creating artifacts as exercised by architects, industrial designers, curriculum developers, composers, etc., rather than to a specific step in a software engineering life-cycle model (located between requirements and implementation). DODEs are computational environments whose value is not restricted to the design of software artifacts. They have been used for the design of software artifacts such as user interfaces, voice dialog systems and Cobol programs, and have served equally well for the conceptual design of material artifacts such 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. They will provide the design support most desirable and most needed and will serve as prototypes for other research efforts moving in the same direction, such as ARPA's research programs in domain-specific software architectures (DSSA) and evolutionary design of complex systems (EDCS).

Historically, software engineering research has been concerned with the transition from specification to implementation ("downstream activities") rather than with the problem of how faithfully specifications really address the problems to be solved ("upstream activities"). Many methodologies and technologies were developed to prevent implementation disasters. The progress made to successfully reduce implementation disasters (e.g., structured programming, information hiding, etc.) allowed an equally relevant problem to surface: how to prevent "design disasters" [Lee 1992] — meaning that a correct implementation with respect to a given specification is of little value if the specification does not adequately address the problem.

Understanding the Problem Is the Problem. The predominant activity in designing complex systems is the participants teaching and instructing each other. Because complex problems require more knowledge than any single person possesses, communication and collaboration among all the involved stakeholders are necessary. Domain experts understand the practice and system designers know the technology. None of these carriers of knowledge can guarantee that their knowledge is superior or more complete compared to other people's knowledge. To overcome this "symmetry of ignorance" [Rittel 1984] as much knowledge from as many stakeholders as possible should be activated with the goal of achieving mutual education and shared understanding.

Integrating Problem Framing and Problem Solving. Design methodologists demonstrate with their work the strong interrelationship between problem framing and problem solving. They argue convincingly that (1) information cannot be gathered meaningfully unless the problem is understood, but one cannot understand the problem without information about it; and (2) professional practice has at least as much to do with defining a problem as with solving a problem. New requirements emerge during development because they cannot be identified until portions of the system have been designed or implemented. The conceptual structures underlying complex software systems are too complicated to be specified accurately in advance and too complex to be built faultlessly. Specification and implementation have to co-evolve, requiring the owners of the problems to be present in the development.

Limitations of Focusing Primarily on Downstream Activities. Many research efforts do not take into account the growing evidence that system requirements are not so much analytically specified as they are collaboratively evolved through an iterative process of consultation between end-users and software developers. For example, CASE tools devise elaborate methods of insuring that software meets its specification but hardly ever question whether there might be something wrong with the specifications themselves. They provide support only after the problem has been solved. A consequence of the thin spread of application knowledge [Curtis, Krasner, Iscoe 1988] is that specification errors often occur when designers do not have sufficient application domain knowledge to interpret the customer's intentions from the requirement statements — a communication breakdown based on a lack of shared understanding. The main objective of formal specifications is that they are "formal," which means that they are manipulable by mathematics and logic and interpretable by computers. As such, these representations are often couched in the language of the computational system. However, such representations are typically foreign and unintelligible to users and get in the way of trying to create a shared understanding among stakeholders.

The Need for Change and Evolution. Software systems model parts of our world. Our world evolves in numerous dimensions as users have new needs, new artifacts appear, new knowledge is discovered, and new ways of doing business are developed. Successful software systems need to evolve. System maintenance and enhancement need to become "first-class design activities." There are numerous fundamental reasons why systems cannot be done "right." Designers are human, and human imagination and knowledge are limited. There is a growing agreement, with empirical data to support it, that the most

critical software problem is the cost of maintenance and evolution [CSTB 1990]. Studies of software costs indicate that about two-thirds of the costs of a large system occur after the system is delivered. Much of this cost is due to the fact that a considerable amount of essential information (such as design rationale) is lost during development and must be reconstructed by the designers who maintain and evolve the system.

In order to create evolvable software systems, the reality of change needs to be accepted explicitly, and increased up-front costs have to be acknowledged and dealt with. The evolution of a software system is driven by breakdowns experienced by the users, which implies that to really support evolutionary processes, users need to be able and willing to change systems and rewarded for doing so. Support for user modification is critical because users of a system are knowledgeable in the application domain and know best which enhancements are needed.

Architectures and Process Models for DODEs. Essential components developed in the context of our research for DODEs are:

* a multi-faceted, domain-independent architecture including the following major components: (1) a construction kit providing a palette of domain concepts; (2) an argumentative hypermedia system containing issues, answers, and arguments about the design domain and the design rationale for a specific application built within the domain; (3) a catalog consisting of a collection of prestored designs that illustrates the space of possible designs in the domain, thereby supporting reuse and case-based reasoning; (4) a specification component supporting the interaction among stakeholders in describing characteristics of the design they have in mind; and (5) a simulation component allowing designers to carry out "what-if" games to simulate various usage scenarios involving the artifact being designed.

* mechanisms for the integration between components including: (1) a specification matcher comparing a specified design profile to a particular artifact design, (2) a critiquing component [Fischer et al. 1991] linking construction and argumentation and providing access to relevant information in the argumentative issue base, (3) an argumentation illustrator helping users to understand the information given in the argumentation base by finding relevant catalog examples that illustrate abstract concepts, and (4) a catalog explorer assisting users in retrieving design examples from the catalog similar to the current construction and specification situation.

* a process model consisting of seeding, evolutionary growth, and reseeding to account for the evolutionary nature of DODEs [Fischer et al. 1994]. A seed for a DODE is created through a participatory design process between software designers and users by incorporating domain-specific knowledge into the domain-independent multi-faceted architecture underlying the design environment. Seeding entails embedding as much knowledge as possible into all components of the architecture. But any amount of design knowledge embedded in design environments will never be complete because real-world situations are complex, unique, uncertain, conflicted, and unstable, and knowledge is tacit (i.e., competent practitioners know more than they can say), implying that additional

knowledge is triggered and activated only by experiencing breakdowns in the context of specific use situations.

Evolutionary growth takes place as users use the seeded environment to undertake specific projects. During these design efforts, new requirements may surface, new components may come into existence, and additional design knowledge not contained in the seed may be articulated. During the evolutionary growth phase, the software designers are not present. Therefore, it is highly desirable that some new design knowledge can be added by the users, requiring computational mechanisms that support end-user modifiability and end-user programming.

Reseeding, a deliberate effort at revision and coordination of information and functionality, brings the software designers back in to collaborate with users to organize, formalize, and generalize knowledge added during the evolutionary growth phases. Organizational concerns play a crucial role in this phase. For example, decisions have to be made as to which of the extensions created in the context of specific design projects should be incorporated in future versions of the generic design environment.

Conclusions

The appeal of the DODE approach lies in its compatibility with an emerging methodology for design; with views of the future as articulated by practicing software engineering experts; with findings of empirical studies; and with the integration of many recent efforts to tackle specific issues in software design (e.g., recording design rationale, supporting case-based approaches, and creating artifact memories). We are further encouraged by the excitement and widespread interest of DODEs and the numerous prototypes being constructed, used, and evaluated in the last few years.

References

[CSTB 1990] *Computer Science and Technology Board, Scaling Up: A Research Agenda for Software Engineering*, Communications of the ACM, pp. 281-293.

[Curtis, Krasner, Iscoe 1988] B. Curtis, H. Krasner and N. Iscoe. *A Field Study of the Software Design Process for Large Systems*, Communications of the ACM, pp. 1268-1287.

[Fischer 1994] G. Fischer. "Domain-Oriented Design Environments", *Automated Software Engineering*, Kluwer Academic Publishers, Boston, MA, pp. 177-203.

[Fischer et al. 1991] G. Fischer et al. *The Role of Critiquing in Cooperative Problem Solving*, ACM Transactions on Information Systems, pp. 123-151.

[Fischer et al. 1994] G. Fischer et al. *Seeding, Evolutionary Growth and Reseeding: Supporting Incremental Development of Design Environments*, Human Factors in Computing Systems, CHI'94 Conference Proceedings (Boston, MA), pp. 292-298.

[Lee 1992] L. Lee, *The Day The Phones Stopped*, Donald I. Fine, Inc., New York.

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

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