13th World Computer Congress 94, Volume 2
K. Brunnstein and E. Raubold (Editors)
Elsevier Science B.V. (North-Holland)

© 1994 IFIP. All rights reserved.

Domain-Oriented Design Environments

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

Abstract. Domain-oriented design environments (DODEs) support the generation and evolution of applications within a particular domain. They reduce the large conceptual distance between problem-domain semantics and software artifacts and they emphasize a human-centered approach facilitating communication about evolving systems among all stakeholders.

This paper discusses the major problems and challenges for future software systems, develops a conceptual framework to address these problems, and describes the major architectural models required for the development of DODEs.

Keyword Codes: D.2.10; H.5.2.; K.8.1
Keywords: Design; User Interfaces; Application Packages

1. Introduction

Design, in the context of this paper, refers to the broad endeavor of creating artifacts (as exercised by architects, industrial designers, curriculum developers, composers, etc., and as defined and characterized, for example, by [16, 14, 3]), 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 they have served equally well for the design of kitchens, lunar habitats, and computer networks. My thesis is that domain-oriented design environments will become as valuable and as ubiquitous in the future as compilers have been in the past, providing much needed and highly desirable design support and serving as prototypes for other research efforts moving in the same direction (e.g., ARPA's research program in *domain-specific software architectures (DSSA)*).

2. Problems and Challenges for Future Software Systems

Understanding the Problem Is the Problem. 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 addressed the problems to be solved (''upstream activities''). The predominant activity in designing complex systems is the participants teaching and instructing each other [2, 9]. Because complex problems require more knowledge than any single person possesses, communication and col-

laboration among all the involved stakeholders are necessary. Domain designers understand the practice and environment developers know the technology. To overcome this "symmetry of ignorance" [13] (i.e., none of these carriers of knowledge can guarantee that their knowledge is superior or more complete compared to other people's knowledge), 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 (e.g., [13, 14]) demonstrate with their work the strong interrelationship between problem framing and problem solving. They argue convincingly that (1) one cannot gather information meaningfully unless one has understood the problem, but one cannot understand the problem without information about it 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 Formal Specifications and CASE Tools. 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 [1]. 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* [2] 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 [12].

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 end-users and get in the way of trying to create a shared understanding between designers and their clients. Ehn [3] notes that *representations for mutual understanding* (such as prototypes, mock-ups, sketches, scenarios, or use situations that can be experienced) are essential "objects-to-think-with" in the design of complex software systems.

The Need for Change. Software systems model parts of our world. Our world evolves in numerous dimensions as new artifacts appear, new knowledge is discovered, and new ways of doing business are developed. Successful software systems need to evolve. Maintaining and enhancing systems need to become "first class design activities." Systems cannot be done "right" for their entire lifetime when they are originally designed, which requires their users to be able to extend and modify them according to their needs.

Understanding People and Their Work. Nothing can be worse than designers who think everyone else is just like them [9]. In the early days of computing, almost all systems were developed and used by computer professionals. Introspection by software designers served as a reasonable source of knowledge at that time, but it has lost its power today for the development of systems in application domains. Research in software engineering in the past has

operated as an overly prescriptive discipline, often postulating a "new human" [16] with interests (e.g., detailed knowledge of low-level computer operation), knowledge (e.g., about work procedures of an application domain), and motivations (e.g., to provide extensive amounts of design rationale, or to deal with formal methods), which had little correspondence with reality.

3. A Theoretical and Conceptual Framework for DODEs

Communication and Coordination. Because designing complex systems is an activity involving many stakeholders, communication and coordination are of crucial importance [9, 7]. By emphasizing design as a collaborative activity, DODEs support three types of collaboration (see Figure 2): (1) collaboration between domain-oriented designers and clients, (2) collaboration between domain designers and design environment developers, and (3) long-term indirect collaboration among designers (creating a virtual collaboration between past, present, and future designers). Design environments provide representations that serve as representations for mutual understanding [3] and therefore help increase the shared context [12] necessary for collaboration.

Domain-Orientation. In a conventional, domain-independent software environment, designers who produce new software artifacts typically have to start with general programming constructs and methodologies [15]. This forces them to focus on the raw materials necessary to implement a solution rather than to try to understand the problem. Design environments need to support *human problem-domain communication* by providing computational environments that model the basic abstractions of a domain (as pursued in efforts in domain modeling). DODEs give designers the feeling that they interact with a domain rather than with low-level computer abstractions. Domain-orientation allows humans to take both the content and context of a problem into account, whereas the strength of formal representations is their independence of specific domains to make domain-independent reasoning methods applicable [11].

Modern application needs are not satisfied by traditional programming languages, which evolved in response to system programming needs. More emphasis should be put on the creation of computational environments that fit the needs of professionals of other disciplines outside the computer science community. The chief risks of using ideas from programming language design and formal specification techniques are in succumbing to the temptations of excess generality and in assuming that users and domain experts think like software designers. The semantics of DODEs are tuned to specific domains of discourse. This involves support for different kinds of primitive entities, for specification of properties other than computational functionality, and for computational models that match the users' own models.

Evolution. There is growing agreement and supporting empirical data that the most critical software problem is the cost of maintenance and evolution [1]. 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 [5]) is lost during development and must be reconstructed by the designers who maintain and evolve the system.

The evolution of a software system is driven by breakdowns [8] experienced by the users of a system. In order to support evolutionary processes, domain designers need to be able, willing, and motivated to change systems, thereby providing a potential solution to the maintenance

and enhancement problems in software design. Users of a system are knowledgeable in the application domain and know best which enhancements are needed. An end-user modification component supports users in adding enhancements to the system without the help of the system developers. End-user modifiable systems will take away from system developers some of the burden of anticipating all potential uses at the original design time.

4. Domain-Oriented Design Environments

In the last several years, DODEs have been developed for the following domains: user interface design, kitchen design, decision support for water management, computer network design (see Figure 1), voice dialog design, COBOL programming, graphics programming, and lunar habitat design (details are provided in [4]).

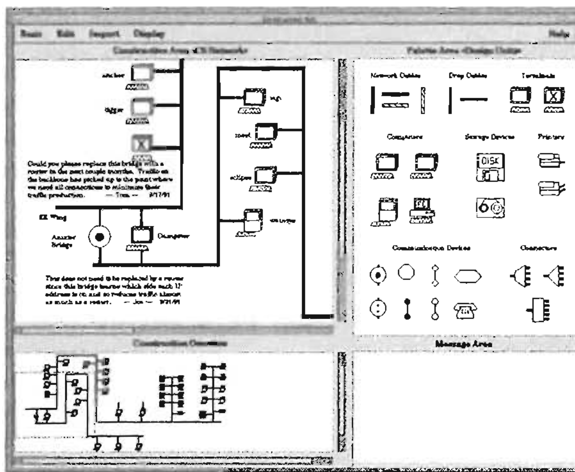

Figure 1: Network — a Design Environment for Computer Network Design

A Domain-Independent Architecture for DODEs. Based on the numerous design efforts creating DODEs as well as on an analysis of the shortcomings of previous efforts, we have developed a domain-independent, multi-faceted architecture to serve as a starting point and organizing framework in the creation of specific DODEs. The major components of this architecture are:

- A *construction kit* is the principal medium for modeling a design. It provides a palette of domain concepts and supports construction using direct manipulation and electronic forms. It supports design by composition.
- An *argumentative hypermedia system* contains issues, answers, and arguments about the design domain and the design rationale for a specific application built within the domain.

- A *catalog* is a collection of prestored designs that illustrate the space of possible designs in the domain and support reuse and case-based reasoning. The primary design activity supported by the catalog is design by modification.
- A *specification component* supports the interaction between clients and designers to describe characteristics of the design they have in mind. The specifications are expected to be modified and augmented during the design process, rather than fully articulated at the beginning. They are used to retrieve design objects from the catalog and to filter information in the hypermedia information space.
- A *simulation component* allows designers to carry out ''what-if'' games to simulate various usage scenarios involving the artifact being designed.

Integration. The multi-faceted architecture derives its essential value from the *integration* of its components. Used individually, the components are unable to achieve their full potential. Used in combination, each component augments the values of the others, forming a synergistic whole. At each stage in the design process, the partial design embedded in the design environment serves as a stimulus to users and suggests what they should attend to next. Links among the components of the architecture are supported by various mechanisms:

- The *CONSTRUCTION-ANALYZER* is a critiquing system [6] that provides access to relevant information in the argumentative issue base. The firing of a critic signals a breakdown to users and provides them with an entry into the exact place in the argumentative hypermedia system where the corresponding argumentation is located.
- The explanation given in argumentation is often highly abstract and very conceptual. Concrete design examples that match the explanation help users to understand the concept. The ARGUMENTATION-ILLUSTRATOR [5] helps users to understand the information given in the argumentative hypermedia by finding a catalog example that illustrates the concept.
- The CATALOG-EXPLORER helps users to search the catalog space according to the task at hand [8]. It retrieves design examples similar to the current construction situation and orders a set of examples by their appropriateness to the current specification.

Seeding, Evolutionary Growth, and Reseeding: A Process Model for DODEs. To account for the evolutionary nature of complex environments that model real-world systems, we have developed a process model for DODE that relies on three major phases: seeding, evolutionary growth, and reseeding (see Figure 2).

A *seed* for a domain-oriented design environment is created through a participatory design process between software designers and domain experts 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 (1) real-world situations are complex, unique, uncertain, conflicted, and instable and (2) 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 domain experts use the seeded environment to undertake specific projects for clients. During these design efforts, new requirements may surface (e.g., the design of a kitchen for people who are blind or in wheelchairs), new components may

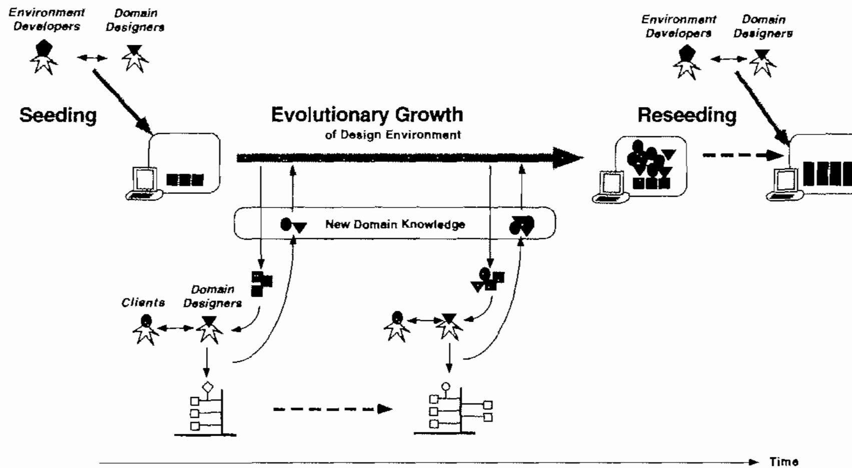

Figure 2: Seeds, Evolutionary Growth, and Reseeding: A Process Model for DODEs

During seeding, environment developers and domain designers collaborate to create a design environment seed that captures an application domain. During evolutionary growth, domain designers (e.g., professional kitchen designers) create specific artifacts. Breakdowns (e.g., the lack of support for specific designs, the ongoing appearance of new components and new knowledge) experienced by the domain designers leads to the addition of new domain knowledge to the seed. In the reseeding phase, environment developers again collaborate with domain designers to organize, formalize, and generalize new knowledge.

come into existence (e.g., microwaves) and additional design knowledge not contained in the seed may be articulated (e.g., that appliances should be against the wall unless we have an island kitchen). During the evolutionary growth phase, the software designers are not present. Therefore it is highly desirable that the new design knowledge can be added by the domain expert requiring computational mechanisms that support end-user modifiability and end-user programming [10].

Reseeding, a deliberate effort at revision and coordination of information and functionality, brings the environment developers back in to collaborate with domain designers 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.

5. Conclusions

The appeal of the DODE approach lies in its compatibility with an emerging methodology for design [3, 14, 16], with views of the future as articulated by practicing software engineering experts [1], with findings of empirical studies [2], 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 [4]). 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.

DODEs (see Figure 3) will lead to further specialization of computer users into environment developers who create (in cooperation with domain experts) the seeds for design environments, and of domain experts who solve problems by exploiting the resources of the design environments. Support for end-user modifiability allows domain experts to extend the functionality of the design environment over time.

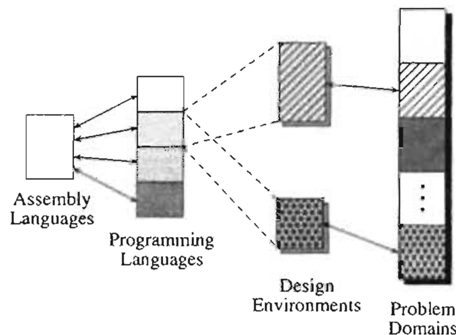

Figure 3: Layered Architectures in DODEs

In the 1950s, programmers had to map problems directly to assembly languages and the assembly programs retained no semantics of the problems to be solved. In the 1960s, general purpose high-level programming languages reduced the transformation distance, which allowed programs to retain some problem semantics and the programming profession was specialized into compiler writers and programmers who developed programs in high-level programming languages.

Design environments reduce the gap between problems and their descriptions as computational artifacts. They make it more likely that software systems correspond to the needs of their users. They facilitate the development and evolution of a particular application within a domain. It is our belief and our hope that they constitute examples of environments within which the majority of future software systems will be developed.

Acknowledgments

This work described is founded on the numerous contributions of the members of the Human-Computer Communication group at the University of Colorado who contributed to the conceptual framework and the systems discussed in this article.

The research was supported in part by (1) the National Science Foundation under grant MDR-9253245, (2) NYNEX Science and Technology Center (White Plains, N.Y.), (3) Software Research Associates, Inc. (Tokyo, Japan) and (4) the Colorado Advanced Software Institute.

References

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

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

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

[4] G. Fischer, "Domain-Oriented Design Environments", *Automated Software Engineering*, Vol. 1, 1994, (in press)

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

[6] G. Fischer, A.C. Lemke, T. Mastaglio, A. Morch, "The Role of Critiquing in Cooperative Problem Solving", *ACM Transactions on Information Systems*, Vol. 9, No. 2, 1991, pp. 123-151.

[7] G. Fischer, J. Grudin, A.C. Lemke, R. McCall, J. Ostwald, B.N. Reeves, F. Shipman, "Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments", *Human Computer Interaction, Special Issue on Computer Supported Cooperative Work,* Vol. 7, No. 3, 1992, pp. 281-314.

[8] G. Fischer, K. Nakakoji, "Beyond the Macho Approach of Artificial Intelligence: Empower Human Designers - Do Not Replace Them", *Knowledge-Based Systems Journal,* Vol. 5, No. 1, 1992, pp. 15-30.

[9] J. Greenbaum, M. Kyng, editors, *Design at Work: Cooperative Design of Computer Systems,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

[10] B.A. Nardi, *A Small Matter of Programming,* The MIT Press, Cambridge, MA, 1993.

[11] D.A. Norman, *Things That Make Us Smart,* Addison-Wesley Publishing Company, Reading, MA, 1993.

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

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

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

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

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