DOMAIN-ORIENTED DESIGN ENVIRONMENTS

Gerhard Fischer

Reprinted from PROCEEDINGS OF THE SEVENTH
KNOWLEDGE-BASED SOFTWARE ENGINEERING CONFERENCE,
McLean, Virginia, September 20-23, 1992

IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

Washington, DC • Los Alamitos • Brussels • Tokyo

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE COMPUTER SOCIETY

Domain-Oriented Design Environments

Gerhard Fischer

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

Abstract

This paper argues that domain-oriented design environments (DODE) provide a complementary goal for the future of software engineering to the approaches pursued with knowledge-based software assistant systems (KBSA). The DODE extends the KBSA framework by emphasizing a human-centered and domain-oriented approach facilitating communication about evolving systems among all stakeholders. The paper briefly discusses the major challenges for software systems, develops a conceptual framework to address these problems, and illustrates the contributions of the KBSA and DODE approaches toward solving these problems.

1. Introduction

Software design (to be understood as the creation and evolution of complex software systems) is a challenging intellectual activity without a "silver bullet" [3] in sight. In order to make progress, we first have to understand what the most pressing problems are. In the next section, I present a brief description of problems (drawn from the literature, from field studies, and from experience). A theoretical and conceptual framework relevant to these problems will be developed in the following section. This framework will be applied to assess the almost 10 year old effort to develop knowledge-based software assistant systems (KBSA) [25]. Domain-oriented design environments (DODE) will be presented as an alternative to the KBSA approach.

2. Framing 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") [2]. Many methodologies and technologies were developed to prevent *implementation disasters* [40]. 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* [40]— meaning that a correct implementation with respect to a given specification is of little value if the specification is wrong to begin with and does not solve the given problem.

Upstream and downstream activities are fundamentally different. They require different groups of people, different methodologies, and different support environments (see Figure 1).

Understanding the Problem Is the Problem. The predominant activity in designing complex systems is the participants' teaching and instructing each other [7]. 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 (they know implicitly *what* the system is supposed to do) and system designers know the technology (they know *how* the system can do it). Based on this "symmetry of ignorance" [37], as much knowledge from as many stakeholders as possible should be activated with the goal to achieve mutual education and shared understanding with respect to the task at hand.

Dertouzous (as reported in [9]) argued that the computer science community should operate less on the supply side (i.e., specifying and creating technology and "throwing the resulting goodies over the fence into the world") and more on the demand side. More emphasis should be put on the creation of computational environments fitting the needs of professionals of other disciplines outside the computer science community. Modern application needs are not satisfied by traditional programming languages that evolved in response to systems programming needs [39, 46].

0-8186-2880-4/92 $03.00 © 1992 IEEE

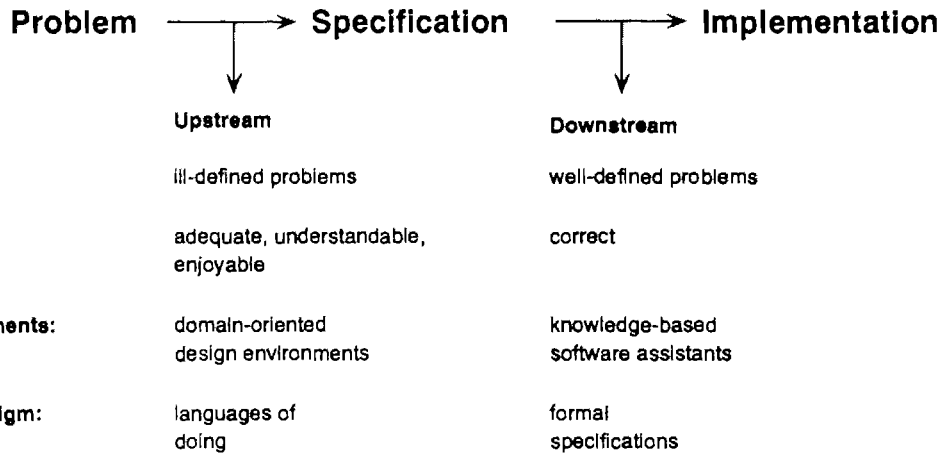

Figure 1: Upstream Versus Downstream Activities

Integrating Problem Setting and Problem Solving. Design methodologists (e.g., [37, 38]) demonstrate with their work the strong interrelationship between problem setting 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 [23], 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 [3]. Specification and implementation have to co-evolve [43], requiring the owners of the problems to be present in the development. If these observations and findings describe the state of affairs adequately, one has to wonder why waterfall models have survived despite the overwhelming evidence that they are not suited for most of today's software problems. One of the reasons for their survival is that traditional management likes them because they provide a model of an orderly process with multiple checkpoints.

Limitation of Formal Specifications. Although there is 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 [6], many research efforts do not take this into account. For example, CASE tools are limited because they devise more elaborate methods of insuring that software meets its specification, hardly ever questioning whether there might be something wrong with the specifications themselves. They provide support after the problem has been solved. A consequence of the *thin spread of application knowledge* [7] is that specification errors often occur when designers do not have sufficient application knowledge to interpret the customer's intentions from the requirement statements — a communication breakdown based on a lack of shared understanding [34].

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. They do not represent *languages of doing* [10] (such as prototypes, mock-ups, sketches, or use situations that can be experienced), whose primary objective is to create mutual understanding among the stakeholders of a problem.

Evolution. Successful systems need to evolve. The need for evolution is based on new and fluctuating requirements [7], new technologies, and the fact that human knowledge is tacit [32] — meaning that we know more than we can say and articulate in the abstract.

Lack of Empirical Research. Nothing can be worse than designers who think everyone else is just like them. 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" [41] 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 in reality. In addition, software design suffers from a lack of detailed analysis of failures and successes of previous systems [31].

Reinventing the Wheel. Software design is a new design discipline compared with other more established disciplines (e.g., architecture, social systems, etc). I claim that software designers can learn a lot by studying these other design disciplines. For example, the limitations and failures of design approaches relying on a strict separation between analysis and synthesis have been recognized in architecture for a long time [37]. A careful analysis of these failures could have saved software engineering the effort expended in finding out that waterfall-type models can at best be an impoverished and oversimplified model of real design activities.

3. A Theoretical and Conceptual Framework

Beyond Automatic Programming: Cooperative Problem Solving. Until recently, many researchers believed (and maybe some still do) that the "*the ultimate goal of artificial intelligence applied to software engineering is automatic programming*" [35]. Rich and Waters [36] modified their position when they argued that the "cocktail party" description of automatic programming is based on a number of faulty assumptions. Rather than "get the human out of the loop," the direction should be "get the computer into the loop" (a goal explicitly articulated for KBSA [25]). Automatic programming in its ultimate sense is not only *not achievable* (because the goals need to be articulated by someone outside the automatic programming system), but it is also *not desirable*, because humans enjoy "doing" and "deciding."

In many situations humans enjoy the process, not just the product. They want to take part in something. This is why they build model trains, why they plan their vacations, and why they design their own kitchens. Automation is a two-sided sword. At one extreme, it can be regarded as a servant, relieving humans of the tedium of low-level operations (e.g., compiling a program, computing the dependency graph between function calls, creating an index for a large document, etc.), freeing them for higher cognitive functions. At the other extreme it can be viewed as reducing the status of humans to "button pushers," stripping work of its meaning and satisfaction. Cooperative problem solving approaches [12] in which computational environments empower, augment and complement human skills and knowledge are a much more desirable goal to pursue than automatic programming. Cooperative problem solving systems raise questions such as (a) which part of the responsibility can or should be exercised by the human and which part by the computer, and (b) how do the human and the intelligent system effectively communicate?.

Communication and Coordination. Because designing complex systems is an activity involving many stakeholders, communication and coordination [16] are of crucial importance, appearing at a number of different levels: (1) between designers and users/clients (where clients do not know what they want), (2) between members of design teams (who might have very different interests), and (3) between designers and their computational knowledge-based design environment.

Domain-Orientation. In a conventional, domain-independent software environment, designers who produce new software artifacts have to start typically with general programming constructs and methodologies. This forces them to focus on the raw material to implement a solution rather than to try to understand the problem. Environments are needed to support not only human-computer communication but *human problem-domain communication* [20]. Human problem-domain communication is facilitated by computational environments that model the basic abstractions of a domain, thereby tuning the semantics of primitives to specific domains of discourse. They give designers the feeling that they interact with a domain rather than with low-level computer abstractions. This allows humans to take into account both the content and context of the problem, whereas the strength of formal representation is that the content and context are irrelevant [30].

Evolution. There is growing agreement (and empirical data to support it) that the most critical software problem is the cost of maintenance and evolution [6]. 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 information (such as design rationale [14]) is lost during development and must be reconstructed by the maintainers and evolvers of the system.

Languages of Doing. The development of complex systems is difficult, not primarily because of the complexity of technical problems, but because of communication and coordination problems, and the need for shared understanding and mutual education about ill-defined problems [26]. Downstream activities are centered around the manipulation and implementation of given specifications, but they do not help to create a shared understanding among all stakeholders. Support environments must serve as *languages of doing* [10] that (1) are familiar to *all* participants, (2) use the practice of the users as a starting point, (3) allow the envisioning of work situations supported by the new systems, and (4) enhance incremental mutual learning and shared understanding among the participants.

4. An Assessment of the "Classical" Model of Knowledge-Based Software Assistant Systems (KBSA)

KBSA (or KBSE, as it was renamed last year to recognize the need to broaden the focus and to rethink some of its goals) is a broad and heterogeneous research effort. Since its original inception in 1983 [25], numerous research activities have been carried out, providing insights into a variety of important software design problems [45]. Rather than enumerating and discussing the achievements of KBSA efforts here (e.g., the ARIES project [27] and the numerous other activities reported in the yearly KBSE conferences), I would like to focus on what I consider shortcomings and questionable goals in order to contribute to an agenda for future research themes.

"Human in the Loop" as a Necessary and Temporary Evil. Many research activities centered around the KBSA concept point in the wrong direction: rather than being *human-centered* (empowering and augmenting all stakeholders in design processes to create more adequate, more understandable, and more enjoyable systems), they are based on the assumption that the "human in the loop" is a necessary and temporary evil (indicating that they still believe in the myths associated with automatic programming [36]).

Understandability of Specifications. Contrary to a basic assumption behind the KBSA effort, I claim that *specification-based descriptions of artifacts have a much narrower scope and are more difficult to develop, maintain, and mutually understand than artifacts supported by languages of doing*. As argued before, formal and decontextualized descriptions may serve well for formal manipulations, but they are not well suited for communication between humans (except for the verification of complicated algorithms and theorems). This claim is supported by W. Wulf [5]: *"I am skeptical that classical mathematics is an appropriate tool for our purposes: witness the fact that most formal specifications are as large, as buggy as, and usually more difficult to understand than the programs they purport to specify. I don't think the problem is to make programming 'more like mathematics'; it's quite the other way around."*

Lack of Domain Orientation. The lack of domain orientation limits (1) the amount of support that a knowledge-based system can provide, and (2) the shared understanding among stakeholders. By necessity, it must focus primarily on downstream activities requiring minimal domain knowledge (e.g., transformations *within* the formal system rather than correspondence of the formal system and the world being modelled).

Lack of Success Models. The assessment given by DeBellis et al. [8] summarizes some additional shortcomings of the KBSA effort, namely lack of evidence for scalability, lack of experiments demonstrating the usability, and insufficient attention to reuse and evolution.

5. Domain-Oriented Design Environments (DODE)

JANUS: An Example. To illustrate DODEs, I will use the JANUS system [21] as an "object-to-think-with." JANUS supports kitchen designers in the development of floorplans. JANUS-CONSTRUCTION (see Figure 2) is the construction kit for the system. The palette of the construction kit contains domain-oriented building blocks such as sinks, stoves, and refrigerators. Designers construct kitchens by obtaining design units from the palette and placing them into the work area. In addition to design by *composition* (using the palette for constructing an artifact from scratch), JANUS-CONSTRUCTION also supports design by *modification* (by modifying existing designs from the catalog in the work area).

The critics in JANUS-CONSTRUCTION [15] identify potential problems in the artifact being designed. Their knowledge about kitchen design includes design principles based on building codes, safety standards, and functional preferences. When a design principle (such as "the length of the work triangle is greater than 23 feet") is violated, a critic will fire and display a critique in the messages pane (Figure 2) identifying a possibly problematic situation (a breakdown), and prompting the designer to reflect on it. The designer has broken a rule of functional preference, perhaps out of ignorance or by a temporary oversight.

Our original assumption was that designers would have no difficulty understanding these critic messages. User experiments with JANUS demonstrated that the short messages the critics present to designers do not reflect the complex reasoning behind the corresponding design issues. To overcome this shortcoming, we initially developed a static explanation component for the critic messages [28] based on the assumption that there is a "right" answer to a problem. But the explanation component proved unable to account for the deliberative nature of design problems. Therefore, argumentation about issues raised by critics must be supported, and argumentation must be integrated into the context of construction. JANUS-ARGUMENTATION (see Figure 3) is the argumentation component of JANUS [14]. It is an argumentative hypermedia system offering a domain-oriented, generic issue base about how to construct kitchens. With JANUS-ARGUMENTATION, designers explore issues, answers, and arguments by navigating through the issue base. The starting point for the navigation is the ar-

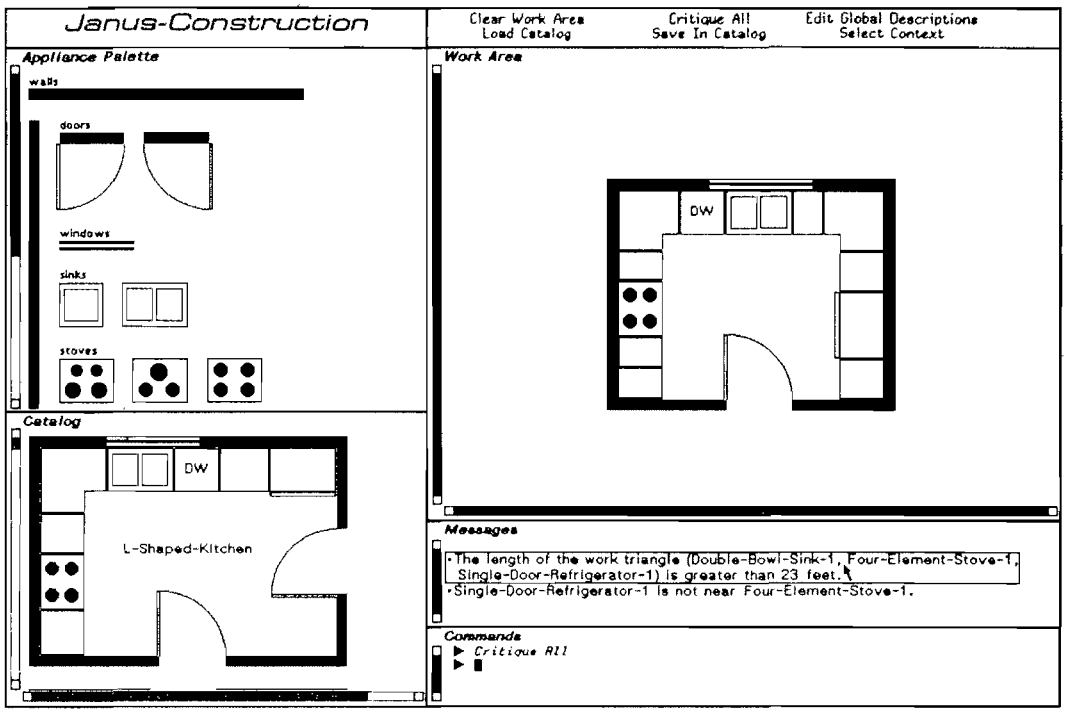

Figure 2: JANUS-CONSTRUCTION: The Work Triangle Critic

JANUS-CONSTRUCTION is the construction part of JANUS. Building blocks (design units) are selected from the *Palette* and moved to desired locations inside the *Work Area*. Designers can reuse and redesign complete floor plans from the *Catalog*. The *Messages* pane displays critic messages automatically after each design change that triggers a critic. Clicking with the mouse on a message activates JANUS-ARGUMENTATION and displays the argumentation related to that message.

gumentative context triggered by a critic message in JANUS-CONSTRUCTION. By combining construction and argumentation, JANUS was developed into an integrated design environment supporting "reflection-in-action" as a fundamental process underlying design activities [38, 22].

A Domain-Independent, Multi-Faceted Architecture for DODE. In addition to JANUS, design environments were developed in different areas throughout the last few years (e.g., user interface design [28], design of decision support system for water management [29], computer network design [16], voice dialog design [42], COBOL programming [1], and graphics programming [17]). From the individual design efforts, we have developed the general architecture shown in Figure 4.

Components. This multifaceted architecture consists of the following five components (Figure 4):

- A *construction kit* (Figure 2) is the principal medium for modeling a design. It provides a palette of domain concepts and supports construction using direct manipulation and electronic forms.
- An *argumentative hypermedia system* (Figure 3) contains issues, answers, and arguments about the design domain.
- A *catalog* (Figure 2) is a collection of prestored designs that illustrate the space of possible designs in the domain and support reuse and case-based reasoning.
- A *specification component* [22] allows 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 to be 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 multifaceted 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

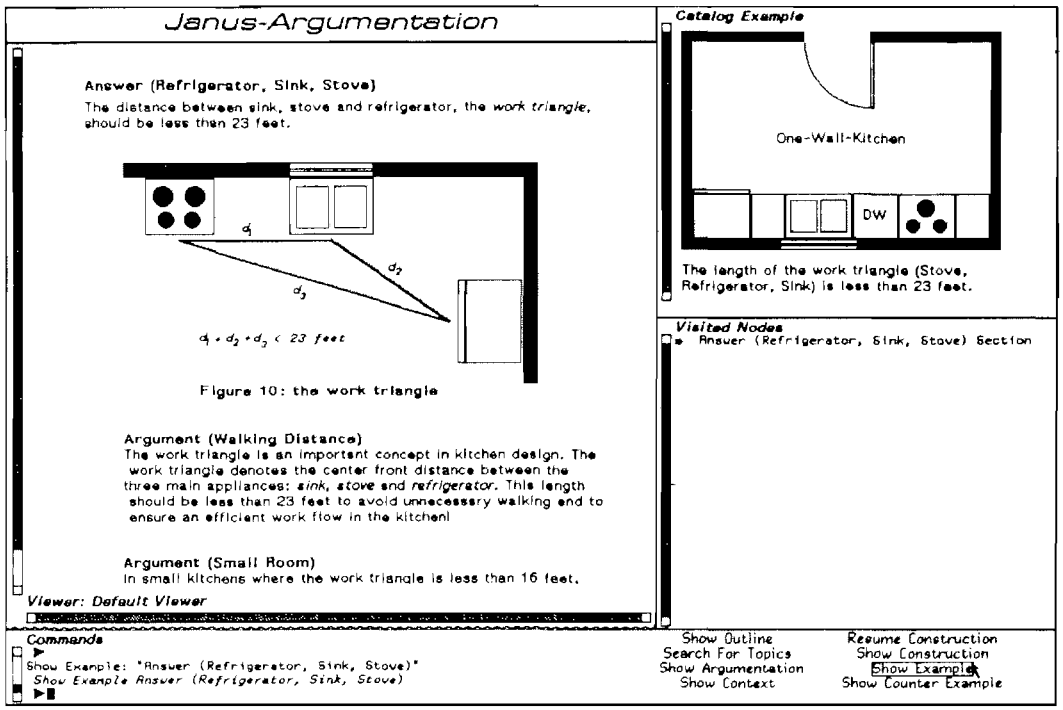

Figure 3: JANUS-ARGUMENTATION: Rationale for the Work Triangle Rule

JANUS-ARGUMENTATION is an argumentative hypermedia system. The *Viewer* pane shows a diagram illustrating the work triangle concept and arguments for and against a work triangle answer. The top right pane shows an example illustrating the answer generated by the ARGUMENTATION-ILLUSTRATOR.

whole. At each stage in the design process, the partial design embedded in the design environment serves as a stimulus to users, suggesting what they should attend to next. Links among the components of the architecture are supported by various mechanisms (see Figure 4):

- *CONSTRUCTION-ANALYZER* is a critiquing system [15] 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.
- *ARGUMENTATION-ILLUSTRATOR.* 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 [14] helps users to understand the information given in the argumentative hypermedia by finding a catalog example that illustrates the concept.
- *CATALOG-EXPLORER.* CATALOG-EXPLORER helps users to search the catalog space according to the task at hand [22]. It retrieves design examples similar to the current construction situation, and orders a set of examples by their appropriateness to the current specification.

Seeding and Evolution. Collaborating domain professionals and software designers need to seed the domain-independent multifacted architecture to create a DODE [17]. Seeding entails embedding as much knowledge as possible into *all* components of the design environment. But design knowledge as 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 [32]), implying that additional knowledge is triggered and activated by situations and breakdowns. These observations require computational mechanisms in support of end-user modifiability [18], end-user programming [24], and programmable applications [11]. The end-user modifiability of JANUS allows users to introduce new design objects, new critiquing rules, and new kitchen designs that fit the needs of different user groups (e.g., blind persons or persons in a wheelchair).

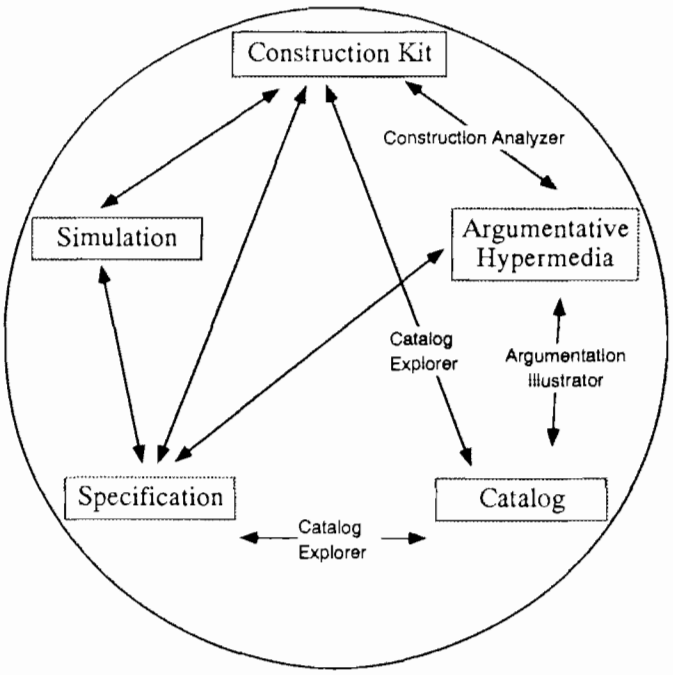

Figure 4: A Multifaceted Architecture

The components of the multifaceted architecture. The links between the components are crucial for exploiting the synergy of the integration.

6. Assessment of KBSA versus DODE

Figure 5 presents a high-level comparison between KBSA and DODE (along the dimensions outlined in the earlier sections).

Current Limitations and Research Issue for DODE. I have argued before that the KBSA approach lacks convincing success models. Obviously, a similar claim can be raised for the DODE approach. The appeal of the DODE approach lies in its compatibility with an emerging methodology for design [4, 10, 38, 41], with views of the future as articulated by practicing software engineering experts [6], with reflections about the myth of automatic programming [36], with findings of empirical studies [7], and with the *integration* of many recent efforts tackling specific issues in software design (e.g., recording design rationale [14], supporting case-based reasoning [33], creating artifact memories [44], and so forth). 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.

Believing DODEs are the way to go, numerous research issues are raised. Creating seeds for a variety of different domains will require substantial resources and the willingness of people from different disciplines to collaborate. Evolving the seeds over time will require more involvement of users and different qualifications, as well as different organizational commitments.

By being high-functionality systems, DODEs create a tool mastery burden. Our experience has shown that the costs of learning a programming language are modest compared to those of learning a full-fledged design environment. New tools (e.g., support for a location/-comprehension/modification cycle [19], critics [15], and support mechanisms for learning on demand [13]) are needed to address these problems.

Redefining the Roles of High-Tech Scribes. There are numerous reasons that a DODE approach will not be readily accepted. Software engineers often have difficulties with the idea that they do not create "universal solutions" that make everyone happy. They have difficulties in sacrificing generality for increased domain-specific support. DODEs replace the clean and controllable waterfall model with a much more interactive situation in which the search for "correct" solutions is limited to downstream activities.

DODE (see Figure 6) will lead to further specialization of computer users into knowledge engineers who create (in cooperation with domain workers) the seeds for design environments, and of domain workers who solve problems by exploiting the resources of the design environments [24]. Support for end-user modifiability allows domain workers to extend the functionality of the

	KBSA	DODE
emphasis	downstream	upstream
	human "in the loop" (as a necessary evil)	human-centered
	automation	cooperative problem solving
	generic	domain-orientation
primary support	formal specifications	languages of doing (supported by the whole DODE)
methodology	getting it right from the beginning	incremental, evolutionary development driven by breakdowns and collaboration
user groups	designers	all stakeholders
interaction level	human-computer communication	human problem-domain communication

Figure 5: A Comparison between KBSA and DODE

design environment over time [18].

7. Conclusions

In conclusion, I want to briefly summarize the main issue of the "message" derived from a DODE perspective.

Emphasis on Humans Rather than on Automation. Rather than "getting the human out of the loop," we should empower designers and users to create and evolve the artifacts fitting their needs and desires. Human-centered communication and collaboration technologies (such as languages of doing) should assist all stakeholders to create shared knowledge and support mutual education.

A Deeper Understanding of Design. Solving ill-defined problems requires the intertwining of problem framing and problem solving. "Understanding the problem is the problem" — which is impossible without an understanding of the problem domain. The role of domain knowledge is critical. Designers do not reason from first order principles, but they rely on experience with similar problems. Design in use is inevitable in a changing world requiring access or reconstruction of the rationale behind the artifact.

Empirical Foundations Through User Experiments. The times of purely prescriptive design methodologies in software engineering belong to the past. "Arm-chair" design and supply-side computing is not sufficient to solve real-world problems. Software is created in the real world, dealing with real tasks, and involving human beings with different interests, skills, and knowledge. To make future computing systems succeed requires more than concern for technology — it requires concern for human beings, their tasks, and their organizations.

Acknowledgments

I would like to thank Stephen Fickas, Dennis Heimbigner, Lewis Johnson, Peter Selfridge, and Loren Terveen, who helped me with important ideas and criticism. The members of the Human-Computer Communication group at the University of Colorado contributed to the conceptual framework and the systems discussed in this article. The research was supported by the National Science Foundation under grant IRI-9015441, and by grants from the Intelligent Interfaces Group at NYNEX; from Software Research Associates (SRA), Tokyo; and by the Software Designer's Associate (SDA) Consortium, Tokyo.

References

[1] M.E. Atwood, B. Burns, W.D. Gray, A.I. Morch, E.R. Radlinski, A. Turner, "The Grace Integrated Learning Environment—A Progress Report", *Proceedings of the Fourth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE 91)*, ACM, June 1991, pp. 741-745.

[2] L. Belady, "MCC: Planning the Revolution in Software", *IEEE Software*, November 1985, pp. 68-73.

[3] F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.

[4] N. Cross, *Developments in Design Methodology*, John Wiley & Sons, New York, 1984.

[5] Computer Science and Technology Board, *Scaling Up: A Research Agenda for Software Engineering*, National Academy Press, Washington, D.C., 1988.

[6] Computer Science and Technology Board, "Scaling Up:

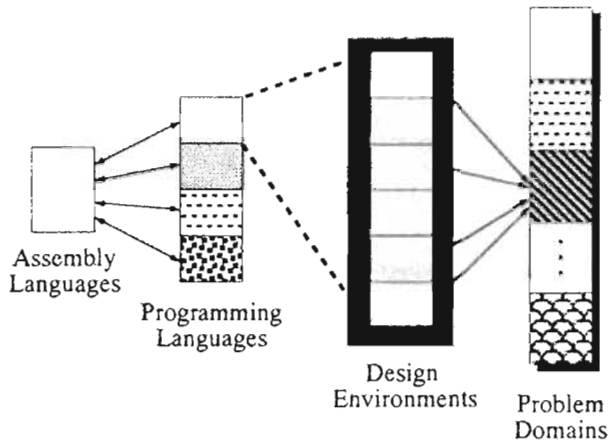

Figure 6: Domain-Oriented Design Environments

In the 1950's, programmers had to map problems directly to assembly languages and the assembly programs retained basically no semantics of the problems to be solved. In the 1960's, general purpose high-level programming languages reduced the transformation distance, allowing programs to retain some problem semantics and the programming profession was specialized into compiler writers and into programmers developing programs in high-level programming languages. Design environments reduce the gap between problems and their descriptions as computational artifacts further by introducing another domain-oriented layer.

A Research Agenda for Software Engineering", *Communications of the ACM*, Vol. 33, No. 3, March 1990, pp. 281-293.

[7] 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.

[8] M. DeBellis, W.C. Sasso, G. Cabral, "Directions for Future KBSA Research", *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September 1991.

[9] P. Denning, "Awakening", *Communications of the ACM*, Vol. 31, No. 11, November 1988, pp. 1254-1255.

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

[11] M. Eisenberg, G. Fischer, "Programmable Design Environments and Design Rationale", *Working Notes of the AAAI 1992 Workshop on Design Rationale Capture and Use*, AAAI, San Jose, CA, July 1992, pp. 81-90.

[12] G. Fischer, "Communications Requirements for Cooperative Problem Solving Systems", *The International Journal of Information Systems (Special Issue on Knowledge Engineering)*, Vol. 15, No. 1, 1990, pp. 21-36.

[13] G. Fischer, "Supporting Learning on Demand with Design Environments", *Proceedings of the International Conference on the Learning Sciences 1991*, Evanston, IL, August 1991, pp. 165-172.

[14] 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.

[15] 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.

[16] 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, (in press)

[17] G. Fischer, A. Girgensohn, K. Nakakoji, D. Redmiles, "Supporting Software Designers with Integrated, Domain-Oriented Design Environments", *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Engineering*, Vol. 18, No. 6, 1992, pp. 511-522.

[18] G. Fischer, A. Girgensohn, "End-User Modifiability in Design Environments", *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*, ACM, New York, April 1990, pp. 183-191.

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

318-328.

[20] G. Fischer, A.C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication", *Human-Computer Interaction*, Vol. 3, No. 3, 1988, pp. 179-222.

[21] G. Fischer, R. McCall, A. Morch, "JANUS: Integrating Hypertext with a Knowledge-Based Design Environment", *Proceedings of Hypertext'89 (Pittsburgh, PA)*, ACM, New York, November 1989, pp. 105-117.

[22] 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.

[23] G. Fischer, B.N. Reeves, "Beyond Intelligent Interfaces: Exploring, Analyzing and Creating Success Models of Cooperative Problem Solving", *Applied Intelligence, Special Issue Intelligent Interfaces*, Vol. 1, 1992, pp. 311-332.

[24] M. Gantt, B.A. Nardi, "Gardeners and Gurus: Patterns of Cooperation Among CAD Users", *Human Factors in Computing Systems, CHI'92 Conference Proceedings (Monterrey, CA)*, ACM, May 1992, pp. 107-117.

[25] C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich, "Report on a Knowledge-Based Software Assistant", Tech. report RADC-TR-83-195, Rome Air Development Center, August 1983, Reprinted in: C.H. Rich, R. Waters (eds): 'Readings in Artificial Intelligence and Software Engineering', Morgan Kaufmann Publishers, Los Altos, CA, pp 377-428, 1986

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

[27] W.L. Johnson, M.S. Feather, D.R. Harris, "The KBSA Requirements/Specification Facet: ARIES", *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September 1991, pp. 53-64.

[28] A.C. Lemke, G. Fischer, "A Cooperative Problem Solving System for User Interface Design", *Proceedings of AAAI-90, Eighth National Conference on Artificial Intelligence*, AAAI Press/The MIT Press, Cambridge, MA, August 1990, pp. 479-484.

[29] A.C. Lemke, S. Gance, "End-User Modifiability in a Water Management Application", Tech. report CU-CS-541-91, Department of Computer Science, University of Colorado, 1991.

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

[31] H. Petroski, *To Engineer Is Human: The Role of Failure in Successful Design,* St. Martin's Press, New York, 1985.

[32] M. Polanyi, *The Tacit Dimension,* Doubleday, Garden City, NY, 1966.

[33] D.F. Redmiles, *From Programming Tasks to Solutions -- Bridging the Gap Through the Explanation of Examples,* PhD dissertation, Department of Computer Science, University of Colorado, 1992.

[34] 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.

[35] C.H. Rich, R. Waters, editors, *Readings in Artificial Intelligence and Software Engineering,* Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[36] C.H. Rich, R.C. Waters, "Automatic Programming: Myths and Prospects", *Computer*, Vol. 21, No. 8, August 1988, pp. 40-51.

[37] 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.

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

[39] 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.

[40] B.A. Sheil, "Power Tools for Programmers", *Datamation*, February 1983, pp. 131-143.

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

[42] T. Sumner, S. Davies, A.C. Lemke, P.Polson, "Iterative Design of a Voice Dialog Design Environment", Tech. report, Department of Computer Science, University of Colorado, 1991.

[43] W.R. Swartout, R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 438-439.

[44] L.G. Terveen, P.G. Selfridge, M.D. Long, "In the Footprints of the Masters: Living Organizational Memory", Tech. report, AT&T Bell Laboratories, 1992.

[45] D.A. White, "The Knowledge-Based Software Assistant: A Program Summary", *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September 1991, pp. vi-xiii.

[46] T. Winograd, "Beyond Programming Languages", *Communications of the ACM*, Vol. 22, No. 7, July 1979, pp. 391-401.