Volume 1, Number 2, June, 1994
ISSN: 0928–8910

AUTOMATED SOFTWARE ENGINEERING

The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering

KLUWER ACADEMIC PUBLISHERS

Automated Software Engineering, 1, 177–203 (1994)

Domain-Oriented Design Environments

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

Abstract. The field of knowledge-based software engineering has been undergoing a shift in emphasis from automatic programming to human augmentation and empowerment. In our research work, we support this shift with an approach that embeds *human-computer cooperative problem-solving tools* into *domain-oriented, knowledge-based design environments*. Domain orientation reduces the large conceptual distance between problem-domain semantics and software artifacts. Integrated environments support the coevolution of specification and construction while allowing designers to access relevant knowledge at each stage within the software development process.

This paper argues that *domain-oriented design environments (DODEs)* are complementary to the approaches pursued with *knowledge-based software assistant systems (KBSAs)*. 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 discusses the major challenges for software systems, develops a conceptual framework to address these problems, illustrates DODE with two examples, and assesses the contributions of the KBSA and DODE approaches toward solving these problems.

Keywords: automatic programming, cooperative problem solving, co-evolution of specification and construction, critiquing, design, domain-oriented design environments, design rationale, end-user modifiability, evolution, FRAMER, formal specifications, JANUS, knowledge-based software assistant, languages of doing, software reuse and redesign, stakeholders, upstream and downstream activities

1. Introduction

Software design is a challenging intellectual activity without a "silver bullet" (Brooks, 1987) in sight. In order to make progress, we first have to understand what the most pressing problems are. The field of knowledge-based software engineering has been undergoing a shift in emphasis from automatic programming to human augmentation and empowerment. A growing number of research efforts are using knowledge-based systems and new communication paradigms to *empower all stakeholders in software design, not to replace them* (stakeholders in a design process are all people who are affected by the design artifact and who are involved in creating and evolving the design artifact). The idea of human augmentation, beginning with Engelbart (Engelbart and English, 1968), has been elaborated in the last 25 years (e.g., Stefik, 1986; Simon, 1986; Hill, 1989; Fischer, 1990; Norman, 1993). It has been applied to the domain of software design through projects such as the the *Programmer's Apprentice* (Waters, 1985), the *Software Designer's Associate* (Kishida et al., 1988), the *Knowledge Base Designer's Assistant* (Schoen, Smith, and Buchanan, 1988), the *Knowledge-Based Software Assistant* (White, 1991), LASSIE (Devanbu et al., 1991), ARIES (Johnson, Feather, and Harris, 1991), and earlier systems described by Barstow, Shrobe, and Sandewall (1984).

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 Simon (1981), Schoen (1983), Ehn (1988), rather than to a specific step in a software engineering life-cycle model (located between requirements and implementation (Royce, 1987)).

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 the design support most desirable and most needed and serving as prototypes for other research efforts moving in the same direction (e.g., ARPA's research program in *domain-specific software architectures*).

In this paper, I first present a brief description of major problems confronting software design (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 (KBSAs) (Green et al., 1983). Domain-oriented design environments (DODEs) will be presented as an alternative to the KBSA approach and will be illustrated by two prototype systems. The paper concludes with an assessment and comparison between KBSAs and DODEs.

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") (Belady, 1985). Many methodologies and technologies were developed to prevent *implementation disasters* (Sheil, 1983). 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* (Sheil, 1983)—meaning that a correct implementation with respect to a given specification is of little value if the specification does not adequately address the problem (Lee, 1992).

Upstream and downstream activities complement each other (Fischer et al., 1991a) and they need to be intertwined (Swartout and Balzer, 1982). But at the same time, they involve different groups of people, and require different methodologies and support environments (see Figure 1). Following are major problems of software design that cannot be solved without taking upstream activities into account.

Understanding the Problem Is the Problem

The predominant activity in designing complex systems is the participants teaching and instructing each other (Curtis, Krasner, and Iscoe, 1988; Greenbaum and Kyng, 1991). Because complex problems require more knowledge than any single person possesses, communication and collaboration among all the involved stakeholders are necessary. Domain

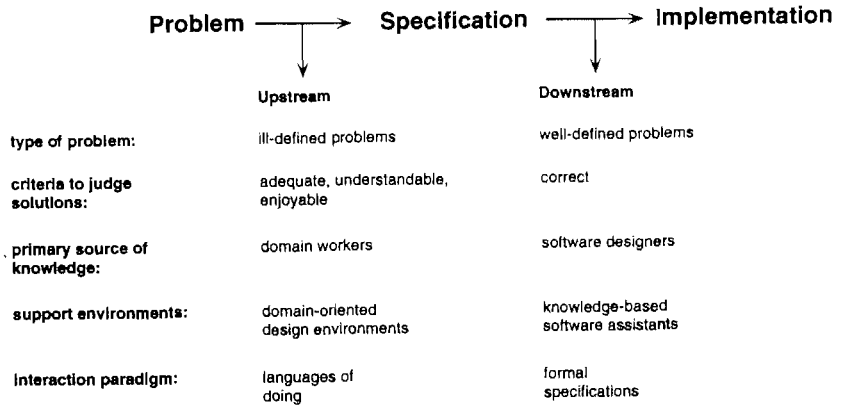

Figure 1. Upstream versus downstream activities.

experts understand the practice and system designers know the technology. To overcome this "symmetry of ignorance" (Rittel, 1984) (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., Rittel, 1984; Schoen, 1983) 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 (Brooks, 1987). Specification and implementation have to co-evolve (Swartout and Balzer, 1982), 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 (Royce, 1987) endure despite the overwhelming evidence that they are not suited for most of today's software problems. Perhaps one reason for their survival is that management likes the evaluative checkpoints possible in serial, orderly process models.

Boehm (1988) analyses different process models and describes the spiral model providing a risk-driven mix of specifying, prototyping and evolutionary development as an attempt to overcome some of the shortcomings of earlier models such as waterfall models. While the spiral model offers general guidelines to take users and domains seriously, it does not emphasize domain-orientation techniques and methods to integrate problem framing and problem solving.

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 (CSTB, 1990). 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, and 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 (Resnick, 1991).

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 (1988) notes that languages of *doing* (such as prototypes, mock-ups, sketches, scenarios, or use situations that can be experienced) are essential "objects-to-think-with" when creating such an understanding.

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," extending system functionality in response to the needs of its users. There are numerous fundamental reasons why systems cannot be done "right." Designers are people, and people's imagination and knowledge are limited.

Understanding People and Their Work

Nothing can be worse than designers who think everyone else is just like them (Greenbaum and Kyng, 1991). 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" (Simon, 1981) 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.

Reinventing the Wheel

Software design is a new design discipline relative to other more established disciplines. I claim that software designers can learn a lot by studying other design disciplines such as architectural design, engineering design, organizational design, musical composition, and writing. For example, the limitations and failures of design approaches that rely on directionality, causality, and a strict separation between analysis and synthesis have been recognized in architecture for a long time (Rittel, 1984). 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. Assessing the successes and failures of other design disciplines does not mean that they have to be taken literally (because software artifacts are different from other artifacts), but that they can be used as an initial framework for software design.

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*" (Rich and Waters, 1986). Rich and Waters (1988) modified their position when they argued that the "cocktail party" description of automatic programming is based on a number of faulty assumptions. Rather than "to get the human out of the loop," the direction should be "to get the computer into the loop" (a goal explicitly articulated for KBSA (Green et al., 1983)).

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 may also be in particular situations not desirable, because humans enjoy "doing" and "deciding." 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.), and thereby freeing them for higher cognitive functions. At the other extreme it can be viewed as reducing the status of humans to "button pushers," and stripping work of its meaning and satisfaction. 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.

Cooperative problem-solving approaches (Fischer, 1990) in which computational environments empower, augment, and complement human skills and knowledge are a more desirable and promising goal to pursue than is 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? Cooperative problem-solving approaches do not deny the power of automation (Billings, 1991), but they focus our concerns on the "right kind of automation" including interaction mechanisms designed for users rather than for programs.

Communication and Coordination

Because designing complex systems is an activity involving many stakeholders, communication and coordination are of crucial importance (Greenbaum and Kyng, 1991; Fischer et al., 1992a). The types of communication and coordination processes that can be differentiated are those between (1) designers and users/clients, (2) members of design teams, and (3) designers and their computational knowledge-based design environment. By emphasizing design as a collaborative activity, domain-oriented design environments support three types of collaboration: (1) collaboration between domain-oriented designers (e.g., professional kitchen designers) and clients (owners of the kitchen to be built), (2) collaboration between domain-oriented designers and design environment builders (software designers), 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 *languages of doing* (Ehn, 1988) and therefore help increase the *shared context* (Resnick, 1991) 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 (Shaw, 1989). 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* (Fischer and Lemke, 1988) by providing computational environments that model the basic abstractions of a domain (as pursued in efforts in domain modeling (Prieto-Diaz and Arango, 1991)). They give designers the feeling that they interact with a domain rather than with low-level computer abstractions. Two such environments, FRAMER (Lemke and Fischer, 1990) and JANUS (Fischer, McCall, and Morch, 1989) are described in later parts of this paper. 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 (Norman, 1993).

Modern application needs are not satisfied by traditional programming languages, which evolved in response to system programming needs (Shaw, 1989; Winograd, 1979). 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 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 (Fischer et al., 1991a; Fischer et al., 1992b)) is lost during development and must be reconstructed by the designers who maintain and evolve the system.

In order to make maintenance and enhancements "first class" activities in the lifetime of an artifact, (1) the reality of change needs to be accepted explicitly and (2) increased up-front costs have to be acknowledged and dealt with. We learned the first point in our work on end-user modifiability (Fischer and Girgensohn, 1990), which demonstrated that there is no way to modify a system without detailed programming knowledge unless modifiability was an explicit goal in the original design of the system. The second point results from the fact that "design for redesign" requires efforts beyond designing for what is desired and known at the moment. It requires that changes be anticipated and structures be created that will support these changes.

The evolution of a software system is driven by breakdowns (Petroski, 1985; Fischer and Nakakoji, 1992) experienced by the users of a system. In order to support evolutionary processes, domain designers need to be able, willing, and rewarded 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 (Henderson and Kyng, 1991).

Languages of Doing

The development of complex systems is difficult, not 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 (Greenbaum and Kyng, 1991). Downstream activities are centered around the manipulation and implementation of given specifications, but they do not help create a shared understanding among all stakeholders. Environments must serve as *languages of doing* (Ehn, 1988) 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.

Communication between clients and designers is difficult because designers and clients use different languages. Explicit representations ground collaborative design by providing a context for communication. These representations (used as languages of doing) help to detect communication breakdowns caused by unfamiliar terminology and tacit background assumptions, and turn breakdowns into opportunities for creating a shared understanding (Fischer and Nakakoji, 1992).

An important component of shared understanding is the *intent* of the collaborators. Understanding intent enhances mutual intelligibility by serving as a resource for assessing the relevance of information within the context of collaboration. In everyday communication between people, intent is often implicitly communicated against a rich background of shared experience and circumstances. Machines, however, have a limited notion of background, and this limits their ability to infer the intent of users (Suchman, 1987).

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

KBSA (Green et al., 1983) was originally envisioned to employ Artificial Intelligence techniques to support all phases of the software development process. The scope of this research effort was broadened in 1991 as the associated conference was renamed Knowledge-Based Software Engineering (KBSE) to recognize the need to broaden the focus and to indicate that the community was moving away from the notion of a super-intelligent computer assistant toward the idea of a *human-computer partnership* (Fischer, 1990; Billings, 1991; Norman, 1993). Rather than to enumerate and discuss the achievements of KBSA efforts here (for examples and discussion, see DeBellis, Sasso, and Cabral (1991); Johnson, Feather, and Harris (1991); and White (1991)), I would like to focus on what I consider shortcomings and questionable goals in order to contribute to an agenda for future research themes.

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 (CSTB, 1988): *"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, which require minimal domain knowledge (e.g., transformations *within* the formal system rather than correspondence between the formal system and the world being modeled).

Lack of Assessment Studies

Software design in general suffers from a lack of detailed analysis of failures and successes of previous systems (Petroski, 1985; Lee, 1992). One of the few assessments of the KBSA effort (DeBellis, Sasso, and Cabral, 1991) summarizes some of the critical shortcomings of past KBSA research, namely lack of evidence for scalability, lack of experiments that demonstrate its usability, and insufficient attention to reuse and evolution.

5. Domain-Oriented Design Environments

In the last several years, numerous DODEs have been developed. Here I use two different environments to illustrate the basic ideas and challenges of DODEs: FRAMER (Lemke and Fischer, 1990) for user interface design, and JANUS (Fischer, McCall, and Morch, 1989; Fischer and Nakakoji, 1992) for kitchen design. Other examples of DODEs are: decision support for water management (Lemke and Gance, 1991), computer network design (Fischer et al., 1992a), voice dialog design (Repenning and Sumner, 1992), COBOL programming (Atwood et al., 1991), graphics programming (Fischer et al., 1992b)), and lunar habitat design (Stahl, 1993).

5.1. FRAMER: A DODE for User Interface Design

FRAMER (Lemke and Fischer, 1990) is a knowledge-based design environment for program frameworks, which are high-level building blocks used for constructing window-based user interfaces (Figure 2). Program frameworks consist of (1) a window frame of nonoverlapping panes, (2) an event loop for processing mouse clicks, (3) keyboard input, and (4) other input events. The program frameworks also manage the update of information displayed on the screen. FRAMER and its architecture is the result of an iterative development process that has gone through three major stages: tool kits, construction kits, and knowledge-based design environments.

Tool Kits

The first stage, tool kits, simply provides domain-oriented building blocks. FRAMER provides designers with components such as windows and menus. Examples of tool kits for the domain of user interface design are Xlib, NextStep, and the Macintosh toolbox. Tool kits enable designers to work in terms of concepts of their domain of expertise rather than at the level of a general-purpose programming language.

Construction Kits

Tool kits provide domain-oriented building blocks, but they do not support the processes of finding and combining the blocks–designers have to know what blocks exist and how they should be used. Construction kits address this problem by providing a *palette* and a *work area* (see Figure 2). The palette displays representations of the building blocks and thus shows what they are and makes them easily accessible. The work area is the principal medium for design and construction in the FRAMER design environment. This is where the designer builds a window layout by assembling building blocks taken from the palette. Examples of user interface construction kits are the Symbolics FrameUp system, MENULAY (Buxton et al., 1983), the NeXt user interface builder, and WIDES and TRIKIT (Fischer and Lemke, 1988).

Design Environments

Knowledge-based design environments address shortcomings that we have found in construction kits. Construction kits support design at a syntactic level only. Our experience with this class of systems has shown that although it is easy to create a functioning interface, creating a *good* interface requires a great deal of additional knowledge that is not provided by construction kits. Design environments provide additional design knowledge through critics, specification sheets, and checklists (see Figure 2).

Critics

Critics (Fickas and Nagarajan, 1988; Fischer et al., 1991b; Fischer et al., 1993) are demons that evaluate the evolving artifact. When the system detects a suboptimal aspect of the artifact, it displays a message that describes the shortcoming in the critic window, which is entitled "Things to take care of" (Figure 2).

Figure 3 shows a typical critic rule. This rule contains knowledge about the relationship of the selected interaction mode and the configuration of window panes in the interface. If the `mouse-and-keyboard` interaction mode is selected, then the rule suggests adding a dialog pane. A `Remedy` action is also defined. Invoking the `Remedy` operation associated with this rule causes the system to add a listener pane at the bottom of the window frame.

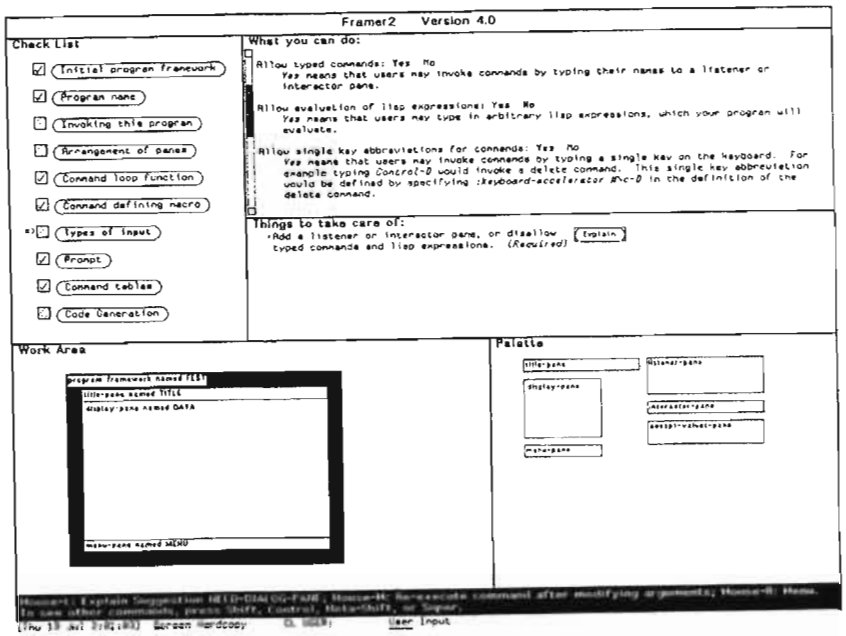

Figure 2. FRAMER: A DODE for user interface design. In the situation shown in the figure, the designer makes a decision about what types of user input should be supported in the interface. The system responds to this decision by displaying a critic message in the critic window entitled "Things to take care of." The critic message identifies a discrepancy between the specification sheet (entitled "What you can do") and the work area. The designer can either modify the window layout in the work area or change the specification sheet.

Specification Sheets

The window layout of an interface has a natural graphical representation as shown in the work area. However, this is not true of all characteristics of an interface. Behavioral characteristics, for instance, must be described in a different way. In the FRAMER system, these other characteristics are described in a symbolic way as fillers in the fields of a specification sheet (see the "What you can do" window in Figure 2). Through the sheet, the system brings design issues and their possible answers to the user's attention and allows users to articulate a partial specification. Associated text explains the significance and consequences of the different design choices.

```
;; A critic rule named need-dialog-pane. The rule applies to program frameworks.
(define-critic-rule need-dialog-pane program-framework
  ;; Applicability condition. This rule is applicable if the interaction mode is
  ;; mouse-and-keyboard. Dialog is conducted either with listener or interactor panes.
  :applicability (equal $interaction-mode mouse-and-keyboard)

  ;; The rule is violated if there is no pane of type dialog-pane in
  ;; the set on inferiors of a program framework.
  :condition (not (exists x (type x dialog-pane)))

  ;; The Remedy operation adds a listener-pane.
  :remedy (let ((pane (make-instance 'listener-pane :x (+ x 20)
                                    :y (+ y 184) :superior self)))
            (add-inferior self pane)
            (display-icon pane))

  ;; Text of the suggestion made to the user if critic is applicable.
  :suggestion "Add a listener or interactor pane, or set the
               interaction mode to mouse-only."

  ;; Text for Praise command.
  :praise "There is a listener or interactor pane."

  ;; Text for Explain command.
  :explanation "Since the interaction mode is mouse-and-keyboard,
                a dialog pane is required for typing in commands.")
```

Figure 3. An example of a critic rule. This is a slightly paraphrased FRAMER critic rule that applies to program frameworks. The rule suggests adding a listener or interactor pane if the interaction mode "mouse-and-keyboard" was specified.

Checklists

The checklist in FRAMER provides an explicit problem decomposition for designers who are unable to decide what steps to take to create a complete functional program framework (thereby supporting the software process (Osterweil, 1987)). The checklist indicates to designers how to decompose the problem of designing a program framework, and it helps to ensure that designers attend to all necessary issues, even if they do not know about them in advance. Each item in the checklist is one subproblem within the overall design process. By selecting a checklist item, designers inform the system of their current focus of attention in the design process. When the designer selects a subproblem in the checklist, the system responds by displaying the corresponding options in the specification sheet shown in the neighboring "What you can do" window and, thus, provides further details about the subproblem. The critics are grouped according to the checklist items. The critic pane always displays exactly those critic messages that are related to the currently selected checklist item. The set of checklist items displayed depends on the designer's previous design decisions.

The system displays only those items that are currently relevant (it is context-sensitive; for example, the prompt item is displayed only if command-based interaction is specified; see Figure 2).

Code Generators

The ultimate goal of user interface design is the generation of an executable program code, and the design activity supported by FRAMER can be viewed as creating a specification for the code. The code generator component of FRAMER is a formal knowledge source that takes care of creating syntactically correct, executable code (the details of the code generation process are discussed by Lemke (1989)).

5.2. JANUS: A DODE for Kitchen Design

JANUS supports kitchen designers in the development of floor plans. JANUS-CONSTRUCTION (see Figure 4) 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 selecting 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 choosing existing designs from the catalog and modifying them in the work area).

The critics in JANUS-CONSTRUCTION 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 should be no greater than 23 feet") is violated, a critic will fire and display a critique in the messages pane (Figure 4) identifying a possibly problematic breakdown situation (Fischer and Nakakoji, 1992), and prompting the designer to reflect on it.

Our original assumption was that designers would have no difficulty understanding these critic messages. User experiments with FRAMER and early versions of 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 (Lemke and Fischer, 1990) based on the assumption that there is a "right" answer to a problem (see Figure 2). But the explanation component proved unable to account for the deliberative nature of design problems (Rittel, 1984). Therefore, argumentation that discusses the pros and cons of issues raised by critics must be supported, and argumentation must be integrated into the context of construction. JANUS-ARGUMENTATION (see Figure 5) is the argumentation component of JANUS (Fischer et al., 1991a). It is an argumentative hypermedia system that offers a domain-oriented 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 argumentative context triggered by a critic message in JANUS-CONSTRUCTION. By combining con-

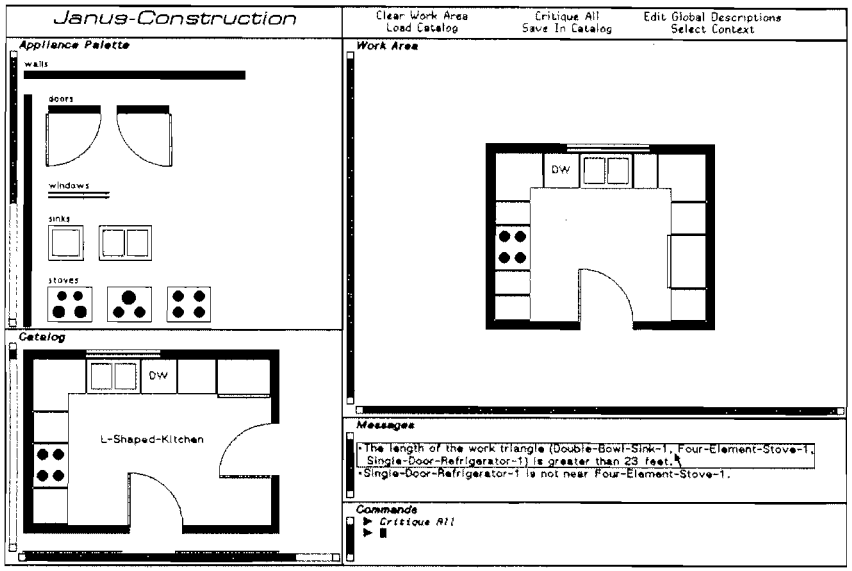

Figure 4. JANUS-CONSTRUCTION: The work triangle critic. JANUS-CONSTRUCTION is the construction part of JANUS. Building blocks (design units) are selected from the Palette and moved to desired locations inside the Work Area. Designers can reuse and redesign complete floor plans from the Catalog. The Messages pane displays critic messages after each design change that triggers a critic. Clicking with the mouse on a message activates JANUS-ARGUMENTATION (see Figure 5) and displays the argumentation related to that message.

struction and argumentation, JANUS was developed into an integrated design environment that supports "reflection-in-action" as a fundamental process underlying design activities (Schoen, 1983; Fischer and Nakakoji, 1992).

5.3. *A Domain-Independent, Multi-Faceted Architecture for DODEs*

Based on the numerous design efforts creating domain-oriented design environments as well as on an analysis of the shortcomings of previous efforts, we have developed the domain-independent architecture shown in Figure 6 to serve as a starting point and organizing framework in the creation of specific DODEs. The individual components as well as the integration mechanisms of this architecture are briefly described below and illustrated with screen images of the JANUS system.

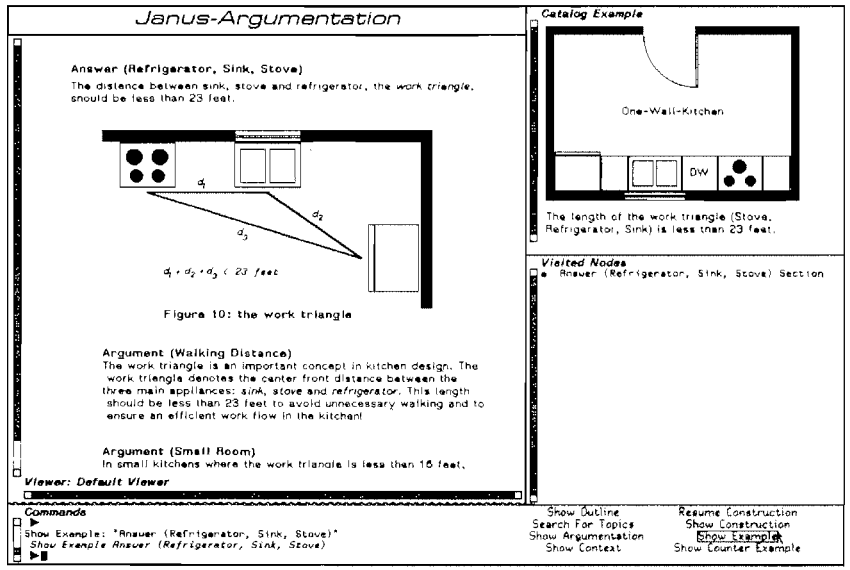

Figure 5. JANUS-ARGUMENTATION: Rationale for the work triangle rule. JANUS-ARGUMENTATION is an argumentative hypermedia system. The Viewer pane shows a diagram of the work triangle concept and arguments for and against a work triangle answer. The top right pane shows an example illustrating the answer in response to the request "Show Example (Refrigerator, Sink, Stove)."

Components

The major components are:

- A *construction kit* (Figure 4) is the principal medium for modeling a design. It provides a palette of domain concepts and supports construction using direct manipulation and electronic forms. The primary design activity supported by it is design by composition.
- An *argumentative hypermedia system* (Figure 5) contains issues, answers, and arguments about the design domain.
- A *catalog* (Figure 4) 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 it is design by modification.
- A *specification component* (Figure 7) supports the interaction between clients and designers to describe characteristics of the design they have in mind (Nakakoji, 1993).

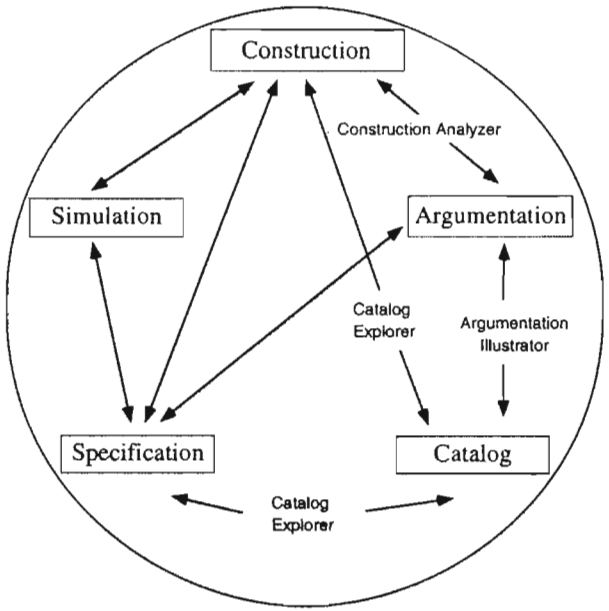

Figure 6. A domain-independent multifaceted architecture.

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.

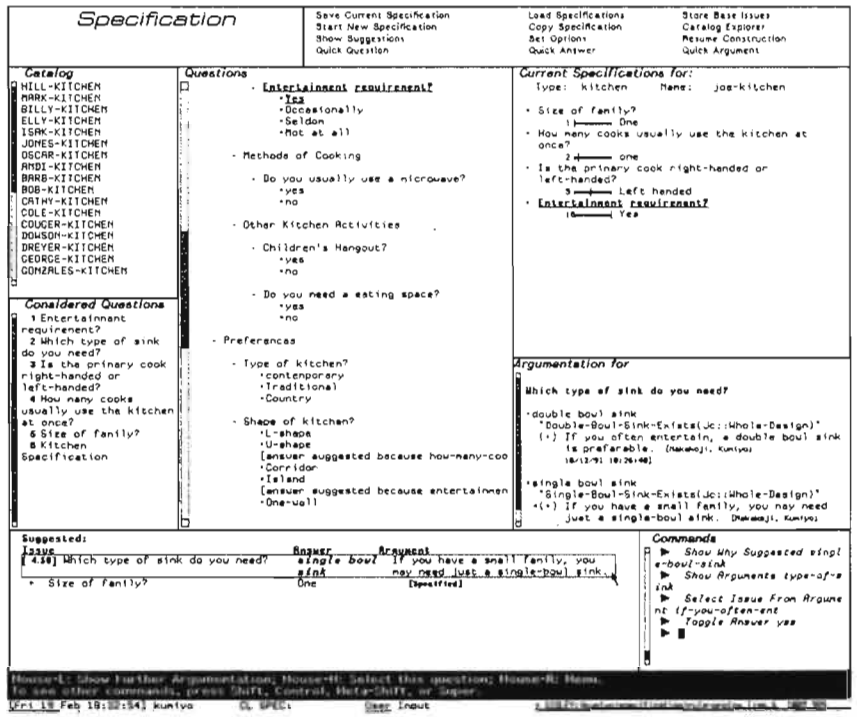

Figure 7. JANUS-SPECIFICATION: Articulation of information about a specific design task. Designers can select answers presented in the Questions window. The summary of currently selected answers appears in the Current Specification window. Each answer is accompanied with a slider (upper right pane) that allows designers to assign a weight representing the relative importance of the answer. Weights are used to prioritize and resolve conflicts between answers (for details see (Nakakoji, 1993)).

Links among the components of the architecture are supported by various mechanisms (see Figure 6):

- The *CONSTRUCTION-ANALYZER* is a critiquing system (Fischer et al., 1991b) 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

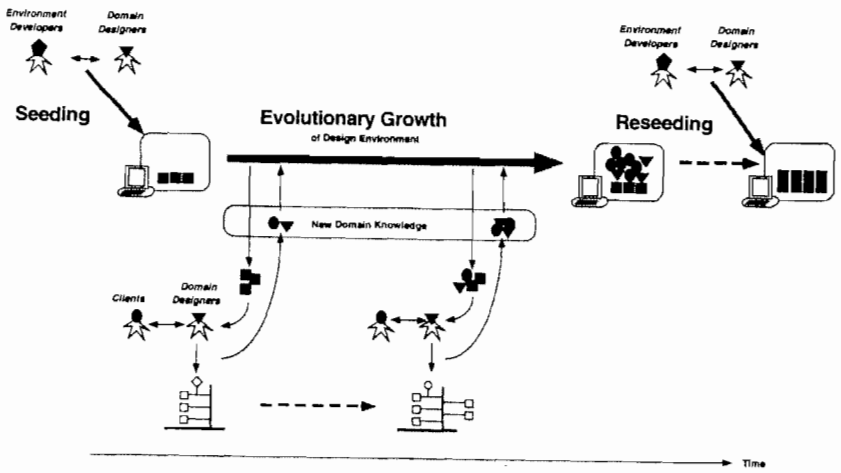

Figure 8. Seeds, evolutionary growth, and reseeding: A process model for DODEs. During seeding, environment developers and domain designers collaborate to create a design environment seed that captures an application domain (e.g., FRAMER or JANUS). During evolutionary growth, domain designers (i.e, professional kitchen designers) create specific artifacts. Breakdowns (e.g., the lack of support for specific designs, the ongoing occurrence 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.

concept. The ARGUMENTATION-ILLUSTRATOR (Fischer et al., 1991a) helps users to understand the information given in the argumentative hypermedia by finding a catalog example that illustrates the concept (see upper right pane in Figure 5).

- The CATALOG-EXPLORER helps users to search the catalog space according to the task at hand (Fischer and Nakakoji, 1992). It retrieves design examples similar to the current construction situation, and orders a set of examples by their appropriateness to the current specification.

5.4. *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 8).

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 (see Figure 6). *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 (Polanyi, 1966)), 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 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 (Fischer and Girgensohn, 1990), and end-user programming (Eisenberg, 1991; Gantt and Nardi, 1992).

Reseeding, a deliberate effort at revision and coordination of information and functionality, brings the software designers back in to collaborate with domain designers to organize, formalize, and generalize knowledge added during the evolutionary growth phases. Organizational concerns (Terveen, Selfridge, and Long, 1993) 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.

After the initial seeding, the use and reseeding phases alternate continuously. Evidence for the adequacy and relevance for this approach can be derived from numerous developments of large-scale software systems that have evolved over time, such as Symbolic's Genera and the X-Window System. In such systems, users develop new techniques and extend the functionality of the system to solve problems that were not anticipated by the system's authors, and distribute them through users' groups. New releases of the system will then incorporate the ideas and code produced by users and found relevant to the community as a whole.

6. Assessment of DODEs

Analogous to the assessment of KBSAs, this section assesses DODEs by comparing and contrasting them with related efforts (software synthesis and requirements engineering) and by describing some of the current limitations and future implications of DODEs. Figure 9 presents a high-level comparison between KBSA and DODE and indicates how they complement each other as KBSAs focus on downstream and DODEs on upstream activities.

Software Synthesis

Research efforts focused around the goal of automatically synthesizing software from higher level specifications and reusable components represent an important idea (compilers rep-

	KBSA	DODE
emphasis	downstream	upstream
	computation-centered	human-centered
	automation	cooperative problem solving
	generic	domain-orientation
primary support	formal specifications	languages of doing (supported by all components of the DODE)
methodology	knowledge acquisition from domain experts	knowledge construction and mutual education driven by breakdowns and collaboration
user groups	primarily software designers	all stakeholders
communication metaphor	human-computer communication	human problem-domain communication

Figure 9. A comparison between KBSA and DODE.

resenting an early success story of these efforts) to increase software productivity. The SINAPSE system (Kant, 1992, 1993) is a system sharing many goals with our systems (e.g., domain-orientation, visualization), but it primarily supports downstream activities (e.g., code generation, optimization), thereby instantiating a number of the goals of the KBSA paradigm and complementing our approach. The different emphasis is mostly due to the fact that the mathematical knowledge is relatively well defined, and formally specified, and the support needed is to empower mathematicians to do their modeling in an environment closer to their world than FORTRAN. New knowledge to SINAPSE is added by the software designer and not by the domain experts (contrary to the view taken by DODEs, as illustrated in Figure 8).

Requirements Engineering

Requirements engineering (Proceedings, 1993) shares many research goals with our efforts on DODEs. It brings together informal system analysis methods as explored in the CSCW community and as used in DODEs in the argumentation component (see Figure 6) with formal methods as explored in KBSA efforts. The *Requirements Apprentice* (Reubenstein and Waters, 1991) explores the formalization phases that bridge the gap between an informal and formal specification, but provides little support for the incremental construction of the informal specifications during the problem-framing process. The *Advisor for Intelligent Reuse (ARI)* (Maiden and Sutcliffe, 1992) assists software designers in identifying and

understanding domain abstractions, but it lacks the linkage between a construction situation and the catalog as provided by the CATALOG-EXPLORER (see Figure 6). The computer-based critic embedded in the *Kate* system (Fickas and Nagarajan, 1988) differs from our critiquing systems as used in the CONSTRUCTION-ANALYZER (Fischer et al., 1991b) in that it analyzes formal specifications rather than construction situations.

Current Limitations and Research Issues for DODEs

The appeal of the DODE approach lies in its compatibility with an emerging methodology for design (Cross, 1984; Ehn, 1988; Schoen, 1983; Simon, 1981), with views of the future as articulated by practicing software engineering experts (CSTB, 1990), with reflections about the myth of automatic programming (Rich and Waters, 1988), with findings of empirical studies (Curtis, Krasner, and Iscoe, 1988), and with the integration of many recent efforts to tackle specific issues in software design (e.g., recording design rationale (Fischer et al., 1991a), supporting case-based reasoning (Redmiles, 1992), creating artifact memories (Terveen, Selfridge, and Long, 1993), 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. Many of our current systems (such as FRAMER and JANUS) rely heavily on a spatial metaphor, but we have also explored other domains in which different properties (such as time in the voice dialog design environment (Repenning and Sumner, 1992) or programming knowledge in the Cobol (Atwood et al., 1991) and graphic design environments (Fischer et al., 1992b)) need to be supported.

DODEs raise numerous research issues. Creating seeds for a variety of different domains will require substantial resources and the willingness of people from different disciplines to collaborate. The necessity to invest in long-term benefits must be taken seriously. Designers who do the work (e.g., providing design rationale) without directly benefiting from their efforts (Fischer et al., 1992a) must be rewarded. Evolving seeds over time will require more involvement of users, a willingness to acquire additional and different qualifications, as well as different organizational commitments (Nardi, 1993).

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 (Fischer et al., 1992b), critics (Fischer et al., 1991b), and support mechanisms for learning on demand (Fischer, 1991)) are needed to address these problems.

New Classes of Computer Users

There are numerous reasons that a DODE approach will not be readily accepted. Software designers 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

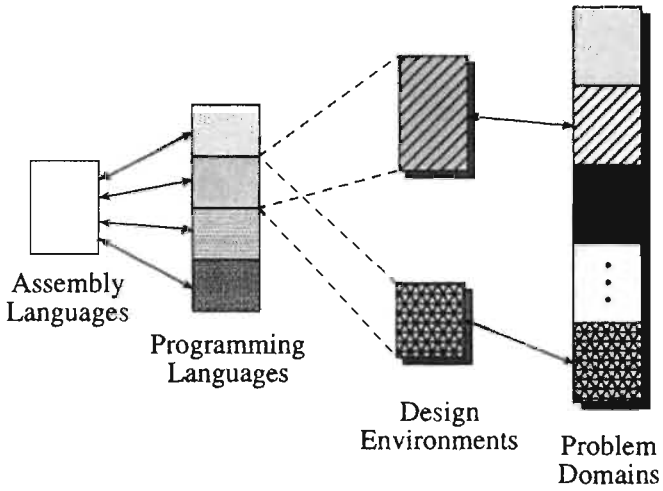

Figure 10. Layered architectures in DODEs. In the 1950s, programmers had to map problems directly to assembly languages and the assembly programs retained basically no semantics of the problems to be solved. In the 1960s, general purpose high-level programming languages reduced the transformation distance, 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 further reduce the gap between problems and their descriptions as computational artifacts by introducing additional, increasingly domain-oriented layers. This approach leads to a layered architecture that underlies all complex systems (Dawkins, 1987).

a much more interactive situation in which the search for "correct" solutions is limited to downstream activities.

DODEs (see Figure 10) 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 (Gantt and Nardi, 1992). Support for end-user modifiability allows domain experts to extend the functionality of the design environment over time (Fischer and Girgensohn, 1990).

7. Conclusions

In conclusion, I want to briefly summarize the main issues 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 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 (achieved by end-user modifiability) is inevitable in a changing world. To make it feasible, end-users require access to the rationale behind the artifact.

Increase in Shared Understanding

Domain-oriented design environments increase shared understanding in three ways: (1) the domain orientation allows a default intent to be assumed, namely, the creation of an artifact in the given domain; (2) the construction situation is accessible and can be "parsed" by the system, providing the system with information about the artifact under construction; and (3) the specification component allows one to explicitly communicate high-level design intentions to the system.

Empirical Foundations Through Assessment Studies in Naturalistic Settings

The times of purely prescriptive design methodologies in software engineering belong to the past. "Arm-chair" design and supply-side computing are not sufficient to solve real-world problems (Thomas and Kellogg, 1989). Software is created in the real world; deals with real tasks; and involves 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

This paper is an extended and revised version of a shorter paper published in the Proceedings of the 7th Annual Knowledge-Based Software Engineering (KBSE-92) Conference. Numerous people have taken the time to discuss and critique the ideas underlying this paper. I would like to thank especially Stephen Fickas, Dennis Heimbigner, Lewis Johnson, Elaine Kant, Peter Selfridge, and Loren Terveen, who helped me with important ideas and criticism. This work would have been impossible without 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. Kumiyo Nakakoji, Jonathan Ostwald, David Redmiles, and Tamara Sumner have provided critical feedback on earlier versions of this article. I would also like to thank the anonymous reviewers of the article, who provided valuable comments and suggestions for improvement.

The research was supported in part by (1) the National Science Foundation under grants No. IRI-9015441 and MDR-9253245, (2) NYNEX Science and Technology Center (White Plains, N.Y.), (3) Software Research Associates, Inc. (Tokyo, Japan), (4) the SDA project, and (5) the Colorado Advanced Software Institute.

References

Atwood, M. E., Burns, B., Gray, W. D., Morch, A. I., Radlinski, E. R., and Turner, A. 1991. *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.

Barstow, D. R., Shrobe, H. E., and Sandewall, E. (eds.) 1984. *Interactive Programming Environments*. New York: McGraw-Hill Book Company.

Belady, L. 1985. MCC: Planning the revolution in software. *IEEE Software*, November: 68–73.

Billings, C. E. 1991. *Human-Centered Aircraft Automation: A Concept and Guidelines*. NASA Technical Memorandum 103885, NASA Ames Research Center, Moffett Field, CA.

Boehm, B. W. 1988. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72.

Brooks, F. P., Jr. 1987. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19.

Buxton, W. A. S., Lamb, M. R., Sherman, D., and Smith, K. C. 1983. Towards a comprehensive user interface management system. *Computer Graphics*, 17(3):35–42.

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

Computer Science and Technology Board. 1988. *Scaling Up: A Research Agenda for Software Engineering*. Washington, DC: National Academy Press.

Computer Science and Technology Board. 1990. Scaling up: A research agenda for software engineering. *Communications of the ACM*, 33(3):281–293.

Curtis, B., Krasner, H., and Iscoe, N. 1988. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287.

Dawkins, R. 1987. *The Blind Watchmaker*. New York: W.W. Norton and Company.

DeBellis, M., Sasso, W. C., and Cabral, G. 1991. Directions for future KBSA research. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September 1991, pp. 138–142.

Devanbu, P., Brachman, R. J., Sefridge, P. G., and Ballard, B. W. 1991. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49.

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

Eisenberg, M. 1991. *Programmable Applications: Interpreter Meets Interface*. Technical Report 1325, Department of Electrical Engineering and Computer Science, MIT.

Engelbart, D. C., and English, W. K. 1968. A research center for augmenting human intellect. In *Proceedings of the AFIPS Fall Joint Computer Conference*, The Thompson Book Company, Washington, D.C., pp. 395–410.

Fickas, S., and Nagarajan, P. 1988. Critiquing software specifications. *IEEE Software*, 5(6):37–47.

Fischer, G. 1990. Communications requirements for cooperative problem solving systems. *The International Journal of Information Systems* (Special Issue on Knowledge Engineering), 15(1):21–36.

Fischer, G. 1991. Supporting learning on demand with design environments. In *Proceedings of the International Conference on the Learning Sciences 1991 (Evanston, IL)*, edited by Lawrence Birnbaum, Association for the Advancement of Computing in Education, Charlottesville, VA, August, pp. 165–172.

Fischer, G., Lemke, A. C., McCall, R., and Morch, A. 1991a. Making argumentation serve design. *Human Computer Interaction*, 6(3–4):393–419.

Fischer, G., Lemke, A. C., Mastaglio, T., and Morch, A. 1991b. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*, 9(2):123–151.

Fischer, G., Grudin, J., Lemke, A. C., McCall, R., Ostwald, J., Reeves, B. N., and Shipman, F. 1992a. Supporting indirect, collaborative design with integrated knowledge-based design environments. *Human Computer Interaction*, Special Issue on Computer Supported Cooperative Work, 7(3):281–314.

Fischer, G., Girgensohn,A., Nakakoji, K., and Redmiles, D. 1992b. Supporting software designers with integrated, domain-oriented design environments. *IEEE Transactions on Software Engineering*, Special Issue on Knowledge Representation and Reasoning in Software Engineering, 18(6):511–522.

Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., and Sumner, T. 1993. Embedding computer-based critics in the contexts of design. *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, ACM, pp. 157–164.

Fischer, G., and Girgensohn, A. 1990. End-user modifiability in design environments. In *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*, ACM, New York, April, pp. 183–191.

Fischer, G., and Lemke, A. C. 1988. Construction kits and design environments: Steps toward human problem-domain communication. *Human-Computer Interaction*, 3(3):179–222.

Fischer, G., McCall, R., and Morch, A. 1989. JANUS: Integrating hypertext with a knowledge-based design environment. In *Proceedings of Hypertext'89 (Pittsburgh, PA)*, ACM, New York, November, pp. 105–117.

Fischer, G., and Nakakoji, K. 1992. Beyond the macho approach of artificial intelligence: Empower human designers—Do not replace them. *Knowledge-Based Systems Journal*, 5(1):15–30.

Gantt, M., and Nardi, B. A. 1992. Gardeners and gurus: Patterns of cooperation among CAD users. In *Human Factors in Computing Systems, CHI'92 Conference Proceedings (Monterrey, CA)*, ACM, May, pp. 107–117.

Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C. 1983. *Report on a Knowledge-Based Software Assistant*. Technical Report RADC-TR-83-195, Rome Air Development Center, August 1983, Reprinted in *Readings in Artificial Intelligence and Software Engineering*, edited by C. H. Rich and R. Waters, pp. 377–428. Los Altos, CA: Morgan Kaufmann Publishers, 1986.

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

Henderson, A., and Kyng, M. 1991. There's no place like home: Continuing design in use. In *Design at Work: Cooperative Design of Computer Systems*, edited by J. Greenbaum and M. Kyng, pp. 219–240. Hillsdale, NJ: Lawrence Erlbaum Associates.

Hill, W. C. 1989. The mind at AI: Horseless carriage to clock. *AI Magazine*, 10(2):29–41.

Johnson, W. L., Feather, M. S., and Harris, D. R. 1991. The KBSA requirements/specification facet: ARIES. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September, pp. 53–64.

Kant, E. 1992. Knowledge-based support for scientific programming. In *Proceedings of the 7th Annual Knowledge-Based Software Engineering (KBSE-92) Conference (McLean, VA)*, IEEE Computer Society Press, Los Alamitos, CA, September, pp. 2–4.

Kant, E. 1993. Synthesis of mathematical modeling software. *IEEE Software*, May: 30–41.

Kishida, K., Katayama, T., Matsuo, M., Miyamoto, I., Ochimizu, K., Saito, N., Sayler, J. H., Torii, K., and Williams, L. G. 1988. SDA: A novel approach to software environment design and construction. In *Proceedings of the 10th International Conference on Software Engineering (Singapore)*, IEEE Computer Society Press, Washington, DC, April, pp. 69–79.

Lee, L. 1992. *The Day The Phones Stopped*. New York: Donald I. Fine.

Lemke, A. C. 1989. *Design Environments for High-Functionality Computer Systems*. Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado, July.

Lemke, A. C., and Fischer, G. 1990. A cooperative problem solving system for user interface design. In *Proceedings of AAAI-90, Eighth National Conference on Artificial Intelligence*, AAAI Press/The MIT Press, Cambridge, MA, August, pp. 479–484.

Lemke, A. C., and Gance, S. 1991. *End-User Modifiability in a Water Management Application*. Technical Report CU-CS-541-91, Department of Computer Science, University of Colorado.

Maiden, N., and Sutcliffe, A. 1992. Domain abstractions in requirements engineering: An exemplar approach? In *Proceedings of the 7th Annual Knowledge-Based Software Engineering (KBSE-92) Conference (McLean, VA)*, IEEE Computer Society Press, Los Alamitos, CA, September, pp. 112–121.

Nakakoji, K. 1993. *Increasing Shared Understanding of a Design Task Between Designers and Design Environments: The Role of a Specification Component.* Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado. Also available as Technical Report CU-CS-651-93.

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

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

Osterweil, L. 1987. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering (Monterey, CA)*, IEEE Computer Society, Washington, DC, March, pp. 2–13.

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

Polanyi, M. 1966. *The Tacit Dimension.* Garden City, NY: Doubleday.

Prieto-Diaz, R., and Arango, G. 1991. *Domain Analysis and Software Systems Modeling.* Los Alamitos, CA: IEEE Computer Society Press.

Proceedings of IEEE International Symposium on Requirements Engineering, 1993. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, January.

Redmiles, D. F. 1992. *From Programming Tasks to Solutions—Bridging the Gap Through the Explanation of Examples.* Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO. Also available as Technical Report CU-CS-629-92.

Repenning, A., and Sumner, T. 1992. Using Agentsheets to create a voice dialog design environment. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, ACM Press, pp. 1199–1207.

Resnick, L. B. 1991. Shared cognition: Thinking as social practice. In *Perspectives on Socially Shared Cognition*, edited by L. B. Resnick, J. M. Levine, and S. D. Teasley, American Psychological Association, Washington, DC, pp. 1–20, ch. 1.

Reubenstein, H. B., and Waters, R. C. 1991. The requirements apprentice: Automated assistance for requirements acquisition. In *IEEE Transactions on Software Engineering*, 17(3):226–240.

Rich, C. H., and Waters, R. (eds.) 1986. *Readings in Artificial Intelligence and Software Engineering.* Los Altos, CA: Morgan Kaufmann Publishers.

Rich, C. H., and Waters, R. C. 1988. Automatic programming: Myths and prospects. *Computer*, 21(8):40–51.

Rittel, H. W. J. 1984. Second-generation design methods. In *Developments in Design Methodology*, edited by N. Cross, pp. 317–327. New York: John Wiley & Sons.

Royce, W. W. 1987. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (Monterey, CA)*, IEEE Computer Society, Washington, DC, pp. 328–338 (reprint of a paper originally published in 1970).

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

Schoen, E., Smith, R. G., and Buchanan, B. G. 1988. Design of knowledge-based systems with a knowledge-based assistant. *IEEE Transactions on Software Engineering*, SE-14(12):1771–1791.

Shaw, M. 1989. Maybe your next programming language shouldn't be a programming language. In *Scaling Up: A Research Agenda for Software Engineering*, edited by the Computer Science and Technology Board, pp. 75–82. Washington, DC: National Academy Press.

Sheil, B. A. 1983. Power tools for programmers. *Datamation*, February: 131–143.

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

Simon, H. A. 1986. Whether software engineering needs to be artificially intelligent. *IEEE Transactions on Software Engineering*, SE-12(7):726–732.

Stahl, G. 1993. *Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design.* Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boulder, CO.

Stefik, M. J. 1986. The next knowledge medium. *AI Magazine*, 7(1):34–46.

Suchman, L. A. 1987. *Plans and Situated Actions.* Cambridge, UK: Cambridge University Press.

Swartout, W. R., and Balzer, R. 1982. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–439.

Terveen, L. G., Selfridge, P. G., and Long, M. D. 1993. From folklore to living design memory. In *Human Factors in Computing Systems, INTERCHI'93 Conference Proceedings*, ACM, April, pp. 15–22.

Thomas, J. C., and Kellogg, W. A. 1989. Minimizing ecological gaps in interface design. In *IEEE Software*, 6 (January): 78–86.

Waters, R. C. 1985. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320.

White, D. A. 1991. The knowledge-based software assistant: A program summary. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, Rome Laboratory, New York, September, pp. vi–xiii.

Winograd, T. 1979. Beyond programming languages. *Communications of the ACM*, 22(7):391–401.

Automated Software Engineering, 1, 205–208 (1994)

Commentary on 'Domain Oriented Design Environments' by Gerhard Fischer

ALISTAIR SUTCLIFFE
City University

The basic message of this paper, that software engineering needs to become more people centric is one which I am sympathetic to. The paper is written in a crusading style and readers unfamiliar with the area may be persuaded that they are witnessing a paradigm shift (in the Kuhn's, 1970, sense) in software engineering from automation to cooperative support. It is this dichotomy which I shall investigate a little further.

Very little attention has been paid to the early stages of software engineering and Fischer quite correctly draws our attention to the fact that the subject, now becoming titled 'requirements engineering', involves complex activities of problem framing, analysis and problem solving. These activities have been familiar to cognitive scientists for many years and several studies have explored problem solving in programming (see Pennington, 1987; Gilmore and Green, 1988), and less frequently in system analysis (Sutcliffe and Maiden, 1991; Guidon and Curtis, 1988). The surprising fact is that little or none of this work has had an impact on the design of CASE tools. The explanation is that technology has outpaced science and only now is the industry waking up to the manifest deficiencies in computer support for software engineering. It is a message which the automated software engineering community would be wise to heed, and Fischer makes this point with reference to CASE and in comparison with support tools for other design domains.

One of the tensions between the automatic programming and nascent design environment tradition is in the nature of the problem. Automatic programming research invariably focuses on small, well-structured problems in engineering and real time type domains. For such problems, formal specification, automatable transformations, and development by refinement may well lead to success. Indeed for safety critical domains formal approaches and automatic programming is not only desirable but essential. However, Fischer reminds us that many problems do not fall into this class. Only too often applications start out as ill defined, requirements are vague, change over time, and are a matter of negotiation. Communication, cooperation and iterative development become more important than formalism. The changing nature of software is not new, (see Lehman, 1971) and forth generation languages can be seen as a partial answer to Fischer's assertion that 'modern application needs' are not served by 'traditional programming languages', So is domain orientation the answer?

The problem with domain centred design is twofold. First there is no sound theory, or even a partial rationale, about what a 'domain' is. Hence domain oriented design environments (DODEs) are build on ad hoc intuition about how large a particular problem is perceived

to be. Note the *perceived*, problems will change in size and nature as they are explored, a point already made in Fischer's paper. Secondly there is the generic trap. If a DODE is constructed for a particular domain then can it be any use in another domain? If not, then this approach leads to a pessimistic conclusion about enhancing software engineering productivity. Every domain will have to have its own purpose-built environment.

Generalisation, and inter alia abstraction, is one of the core concerns of software engineering. This problem has surfaced in reuse research where templates or generic models have been proposed in an attempt to reuse domain knowledge (Reubenstein, 1990; Maiden and Sutcliffe, 1991). However the commercial reality of software reuse is that success has only been found in modest domain specific libraries (Arango et al., 1993). This argues that domain oriented approached may be the way forward but it still leaves open the question of how large or small, general or specific, a domain should be. DODEs may be able to deliver modest evolutionary/ within domain reuse but nothing more. However we should also judge them on other criteria in which they claim strength namely requirements analysis and validation and ones in which they are somewhat silent; the reliability and quality of the designed product.

Fischer advocates 'languages for doing' for design, however, it is not entirely clear what constitutes such as language. Some of the examples appear to be close to visual programming (e.g. FRAMER) while others are constraint based graphical editors (e.g. JANUS) in the Thinglab and ARK tradition (Smith, 1987). Exploration of design ideas could be either restricted by a system which does not allow freedom of action by the designer; alternatively, an overpermissive system could just encourage poor design. The problem here is the lack of any theoretical basis for cooperation. Models of cooperation are still in their infancy in Human Computer Interaction and Knowledge Based Systems, yet such models are necessary to decide what support should be automated in a DODE and furthermore how the dynamic process of cooperation should be managed in a dialogue. At present it appears that cooperation in Fischer's systems takes a safe line of leaving initiative with the user, although we have no way of know how optimal, or sub optimal, this strategy is.

Critique is a key component of DODEs, so design proceeds more by human creation and machine review rather than machine automation and guidance as in the intelligent CASE assistant tradition (e.g. Punchello et al., 1988; Johnson et al., 1991). Unfortunately the penalty of domain specific knowledge means that the value of the critiquer tool is a function of the knowledge acquisition effort in the domain, to say nothing of the knowledge representation and dialogue design. As knowledge acquisition is an acknowledged bottle neck in KBS development, the outlook for intelligent critiquers is pessimistic. Every domain will have to have an exhaustive analysis to find all the principles, rules, guidelines etc for good design and then worse still, collection of 'buggy rules' to detect poor practice. Domain analysis has had a poor track record of price performance payback in reuse (see the DRACO project, Neighbours, 1989), moreover buggy rules have proved hard to collect in Intelligent Tutoring Systems. This raises a further question, how far does a DODE go in its intelligence? Clearly there is a stopping problem as a critiquers could soon become a complete ITS with pedagogical and diagnostic modules.

One riposte which Fischer can make to the domain specific limitation, is that his work produces domain independent architecture. While this is true, the question becomes one

of payback in increased design quality for effort in customising and configuring a generic architecture. To go through the components listed in section 5.3:

- The construction kit; considerable effort will be necessary to turn a 'palette of domain concepts' into support for construction by direct manipulation. A graphical composition grammar will have to be designed and mapped to the underlying semantics of the objects, relationships, etc., in the domain. Work by Sommerville et al. (1987) on generic CASE environments may be helpful here. The construction kit should provide customisable languages for design support.

- An argumentation hypermedia system; this is a good idea but why not use well known systems such as gIBIS?

- A catalogue of prestored designs; another fine idea but what happens when the library scales up? This encounters the retrieval for reuse problem. Experience suggests that beyond 1–2000 designed components, people have problems using libraries without retrieval support, be that by faceted classification schemes or intelligent retrieval engines.

- A specification component; this is probably the most difficult part, models of cooperation are necessary, and the domain knowledge bottleneck makes the critiquer expensive to build. Further problems are how to link argumentation structures to evolving designs, to say nothing of supporting other facets of design known from cognitive studies, such as maintenance of multiple hypotheses (alternative workspaces), and support for working memory (designer's note pads).

- Simulation component; it is essential to be able to animate and run various designs, although can different views be incorporated, how can scripts and scenarios be run against a design? Can scripts of people moving and cooking in kitchens be run in JANUS simulations?

While the concept looks attractive, considerable customisation will be necessary to build a DODE from such a toolkit. Furthermore there appear to be several fairly fundamental research questions that have to be answered on the way.

DODEs are envisaged as evolving environments (section 5.4) with gradual acquisition of domain and design knowledge. Unfortunately this brings with it the danger of integrity maintenance. Updating knowledge bases is a hazardous business even with good controls. One designer's rule may well clash with another's, and constructing an environment which can rigorously check for rule inconsistencies is difficult especially as the size of the domain knowledge base grows.

If DODEs can be delivered within the bounds of reasonable effort, there is good reason to expect that designed products may have a better quality. Requirements capture and validation should be improved. However, there is little in Fischer's paper about improvement in reliability of software products. Surely DODEs should attempt to ensure that the resulting designs are specified so that the resulting code is reliable and even maintainable? This implies that DODEs need to have specification formalisms and code generation by automatic programming built into them, a synthesis indicated in the paper. The link that is not made by Fischer is that formalism itself enables reasoning about specifications and hence can feed

into design critiques. Formal software engineering tools and design environments may have a symbiotic relationship rather than the more separate 'upper and lower' CASE model.

Finally to return to the crusade. Are DODEs are paradigm shift from formal software engineering tools and indeed pragmatic structured methods? I will not venture a verdict, but I do suggest that the conception of cooperation, designer/user centricity and domain orientation is growing in a number of research communities including automatic programming and formal methods. Design environments may synthesise research on simulation, cooperative assistants, domain analysis and reuse into a new coherent direction for CASE; although the benefits of formal software engineering need to be integrated into the vision. Gerhard Fischer has made a good case for this vision, although some of the problems inherent in realising design environments have been finessed. Considerable theoretical and applied research is necessary to turn DODEs from application specific demonstrators into a general engineering technology.

References

Arango, G., Schoen, E., and Pettengill, R. 1993. Design as evolution and reuse. In *Proceedings of Advances in Software Reuse, Proceedings of 2nd International Workshop in Software Reuse*, edited by R. Preito Diaz and W. B. Frakes. IEEE Computer Society Press.

Gilmore, D. J., and Green, T. R. G. 1988. Programming plans and programming experience. *The Quarterly Journal of Experimental Psychology*, 40A:423–442.

Guindon, R., and Curtis, B. 1988. Control of cognitive processes during software design: What tools are needed? In *Proceedings of CHI '88 Conference: Human Factors in Computer Systems*, edited by E. Soloway, D. Frye, and S. B. Sheppard, pp. 263–269. ACM Press.

Johnson, W. L., Feather, M. S., and Harris, D. R. 1991. The KBSA requirements/specification facet: ARIES. In *Proceedings of 6th annual Knowledge based software engineering conference (KBSE-91)*, Syracuse NY, 53–64.

Kuhn, T., 1970. *The Structure of Scientific Revolutions*, 2nd, Chicago.

Lehman, M. M., 1991. Software Engineering, the software process and their support tools. *Software Engineering Journal*, Vol. 6(5), pp. 243–258.

Maiden, N. A. M., and Sutcliffe, A. G. 1992. Exploiting reusable specification through analogy. *Communications of the ACM*, 35(4):55–64.

Neighbours, J. M. 1989. Draco, A method for engineering reusable software systems. In *Software Reusability*, edited by T. Biggerstaff and A. Perlis. ACM Press.

Pennington, N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341.

Punchello, P. P., Torrigiani, P., Pietri, F., Burion, R., Cardile, B., and Conit, M. 1988. ASPIS, a knowledge based CASE environment. *IEEE Software*, March: 58–65.

Reubenstein, H. B. 1990. *Automated Acquisition of Evolving Informal Descriptions*. Ph.D. Dissertation (A.I.T.R. No. 1205), Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Sommerville, I., Welland, R., and Beer, S. 1987. Describing software design methodologies. *Computer Journal*, 30(2):128–133.

Smith, R. 1987. Experiences with the alternate reality kit: An example of the tension between literalism and magic. In *Human Computer Interaction, Proceedings of CHI-87*, edited by J. M. Carroll and P. Tanner, pp. 61–68. ACM Press.

Sutcliffe, A. G., and Maiden, N. A. M. 1991. Analogical software reuse: Empirical investigations of analogy based reuse and software engineering practices. *Acta Psychologica*, 78(1–3):173–197.

Automated Software Engineering, 1, 209–213 (1994)

Birds of a Feather: The DODE and Domain-Specific Software Synthesis Systems

DOROTHY E. SETLIFF
University of Pittsburgh

1. Introduction

Fischer, in his paper "Domain-Oriented Design Environments", introduces DODEs as a new design process. The fundamental characteristics of the DODE are the incorporation of domain-specific knowledge, the separation into synthesis, analysis, and simulation design components, and an emphasis on stakeholder and synthesis cooperation. Fischer contrasts the DODE against current knowledge based software engineering techniques, which he claims emphasizes the replacement of humans in design [Section 1]. Fischer claims that the ultimate goal of these techniques is automatic programming [Section 1]. Fischer characterizes automatic programming as being 'unachievable' [Section 3] and the antithesis of the DODE design process.

This author disagrees with Fischer's characterization of automatic programming, and believes that Fischer's DODE design process is virtually identical to that seen in current automatic programming/software synthesis design systems. While Fischer does not provide much in the way of algorithm or technique suggestions to implement the DODE design process, this author believes that current software synthesis techniques can be used to instantiate much of the DODE design process. The remainder of this paper presents a more current definition of automatic programming and software synthesis and investigates the similarity between current software synthesis systems and Fischer's DODE design process. Given this similarity, this author believes that the crusade to empower, rather than replace, humans is already a point in fact and that the DODE design process is not a totally new idea. The organization of ideas presented in Fischer's paper does cover in one place the major areas of research needed to fully implement the DODE. Current software synthesis systems represent steps in that direction.

2. Automatic Programming Today

Automatic programming, more currently called software synthesis, focuses on the tools and techniques needed to synthesize a specific type of design object, namely, software. Software has proven to be quite difficult to design automatically. This is because software as a design object is so easily modifiable and quite often the 'glue' tying together a multitude of different design objects (e.g., hardware) (Royce, 1993), Early automatic programming efforts centered on generic (i.e., domain-independent) techniques and on the software coding process (Biermann, 1976; Schonberg et al., 1981), rather than on the software design

process. Several efforts included domain-specific knowledge (Barstow, 1979), but these resulted in application-specific code generators, and were without a good abstraction of the design architecture. Unfortunately, the limitations of these early efforts soured many researchers on the applicability and likely success of software synthesis. It is the limitations of these early efforts that drives Fischer's characterization of automatic programming.

Fortunately, software synthesis has progressed past these first beginnings. The realization that design, defined by Dym as "the act of translating requirements into specifications and constraints" (Dym, 1993), is the real problem, not implementation. Thus, software synthesis turned its focus to software design issues (Setliff et al., 1993). The recognition that the domain affects design and vice versa is one that Fischer specifically incorporates in his DODEs and can be seen in current software synthesis and engineering systems (Lowry, 1991).

One design field that has shown startling synthesis success is VLSI CAD design. The combination of domain-independent synthesis algorithms and domain-specific knowledge successfully synthesizes VLSI CAD designs. Three characteristics serve to make VLSI CAD highly synthesizable: the specification of a specific target solution architecture, the restriction to a set of known design components, and high reusability. Current software synthesis approaches seek to mirror the success of VLSI CAD design synthesis by making the same set of design style restrictions and using the same combination of domain-independent synthesis algorithms and domain-specific knowledge (Setliff and Rutenbar, 1992; Smith and Setliff, 1993; Keller and Rimon, 1992).

The success of domain-specific software synthesis has not been noted in many circles. Software synthesis has successfully produced systems targeting a specific design architecture in a given domain (Smith and Setliff, 1993; Jullig and Pressburger, 1993; Eriksson and Musen, 1993; Kant, 1993; Abbott et al., 1993). This allows for greater domain breadth and insights into how to transfer this success into other related domains. Work is continuing on design style abstractions so that synthesis can evaluate the relative merits of different design architecture styles.

Fischer is incorrect in believing that software synthesis seeks to replace the human in design [Section 3]. Rather, software synthesis seeks to tailor the design process (consisting of synthesis, analysis, and simulation activities) according to what the designer believes is an essential human design activity in accordance with the maturity of the target domain. This is identical to the design needs that Fischer claims are met by DODEs. It is when the human performs repetitive actions that the human becomes a 'button-pusher' [Section 3], not when automation is provided. Synthesis focuses on activities that the human has no real desire to participate in because they are repetitive, even when those activities are software design activities. The most likely repetitive design activities are in well-known domains. While most domains are not well-known at this point, allowing synthesis to perform design activities allows the designer to quickly evaluate the repercussions of different options. Humans prefer to evaluate the effectiveness of decisions. It is this facility that synthesis provides.

3. A Comparison of Software Synthesis Systems and the DODE

It is this author's contention that both Fischer's DODE and current software synthesis systems seek to empower the human, incorporate domain knowledge where necessary, and separate the design process into synthesis, analysis and simulation components. Fischer lists five main components in the generic DODE architecture. Iteration is made explicit through an integration of these components. Each of these five main components has a corresponding component in most software synthesis systems. The following list summarizes Fischer's DODE component description, describes the corresponding software synthesis system component, and highlights their similarities and differences.

- Construction kit:

  Fischer describes this component as a 'palette of domain concepts' that supports 'design by composition.' This component supports synthesis activities. A number of software synthesis architectures duplicate the functionality of this component by incorporating templates of various abstraction levels (Graves et al., 1992; Kant, 1993; Keller and Rimon, 1992). These templates act as design knowledge supporting synthesis operations. Keller and Rimon (1992) use templates of mathematical behaviors to derive complex mathematical design functions. Setliff and Rutenbar (1992) use design templates much as does Janus [Section 5.1] to modify initial designs to meet the evolving routing specification. Software synthesis architectures demonstrate the utility of 'design by composition', especially in rapidly changing domains (Setliff and Rutenbar, 1992).

- Argumentative hypermedia system:

  Fischer expounds the use of visual techniques to provide analysis functions. This component acts as an enabling user interface for database technologies. The goal of this component is provide a domain-specific user interface while supporting explanation-based design (i.e., human/computer cooperation). While no software synthesis system has explicitly detailed the need for this type of visual interaction, Kant (1993) has explored the use of history keeping to explain design decisions. Jullig (1993) supports combined explanation and simulation activities. Abbott et al. (1993) provides a visual domain-specific specification user interface. Thus, current software synthesis architectures support parts of this component. Fischer makes a good point that user interactions should be in terms of the domain. Successful software synthesis architectures have domain-specific user interfaces.

- Catalog:

  Fischer describes the catalog as 'a collection of pre-stored designs that illustrate the space of possible designs' and as a support for 'design for modification' [Section 5]. This component is primarily a database supporting the activities in the construction kit. Indeed, in most software synthesis architectures, this component is contained within the construction kit (using Fischer's terminology) (Setliff and Rutenbar, 1992; Smith and Setliff, 1993; Kant, 1993; Keller and Rimon, 1992). Thus, the combination of the catalog and construction kit (and in part the specification component below) perform synthesis and analysis activities in software synthesis architectures.

- Specification:

  Fischer argues for domain-specific specification interfaces and languages (languages for doing). The specification is expected to be modified as a function of performing design. Domain-specific software architectures have embraced domain-specific specifications. Keller and Rimon (1992) use visual data flow graphs to succinctly capture physical to mathematical relationships. Smith and Setliff (1993) use simple tables common in use by systems analysts. Abbot et al. (1993) use graphical editors to capture synthesis for parallel process. Software synthesis architectures embrace the use of domain-specific user interfaces and typically incorporate Fischer's specification component into Fischer's argumentative hypermedia system component.

- Simulation:

  Fischer specifically incorporates the necessity of simulation to play 'what if' games during the design process. Simulation is supported by Jullig (1993), Abbott et al. (1993), and Keller and Rimon (1992). Simulation is provided at different abstraction levels: design, functionality, and implementation. Effective simulation requires efficient analysis of the end-result, software. Software synthesis techniques are uniquely suited for effective simulation and can be used in DODEs to provide analysis of the design space.

Current software synthesis architectures, while not split precisely into Fischer's five components, generally support the 'gist' of Fischer's design process. Software synthesis does have an emphasis on abstraction levels (due to a philosophical difference on what is automatable) not present in Fischer's discussion of DODEs.

4. Conclusions

Software synthesis techniques provide most, but not all, of the functionality in Fischer's DODE. Software synthesis is achievable and is best at the automation of repetitive activities, even when these activities are design. Software synthesis performs analysis and synthesis activities automatically in well-known domains, while in less well-known domains, software synthesis supports simulation activities for human cooperation. Software synthesis techniques support human cooperation by effectively evaluating different options within the design space. Current software synthesis architectures are well on their way towards instantiating Fischer's DODE design process.

References

Abbott, B., Bapty, T., Biegl, C., Karsai, G., and Sztipanovits, J. 1993. Model-based software synthesis. *IEEE Software*, May: 42–52. Los Alamitos, CA: IEEE Computer Society.

Barstow, D. 1979. An experiment in knowledge based automatic programming. *Artificial Intelligence*, 12:73–119.

Biermann, A. W. 1976. Approaches to automatic programming. In *Advances in Computers*, edited by M. Rubin and M. C. Yovits, Vol. 15, pp. 1–63. New York: Academic Press.

Biggerstaff, T. J., and Perlis, A. J. 1989. Software reusability. In *Concepts and Models*, Vol. I. ACM Press.

Dym, C. 1993. *Teaching Design*. Presentation at NSF Workshop on the Freshman Engineering Experience, Colorado State University, July.

Eriksson, H., and Musen, M. 1993. Metatools for knowledge acquisition. *IEEE Software*, May: 23–29. Los Alamitos, CA: IEEE Computer Society.

Graves, H., Louie, J., and Mullen, T. 1992. A code synthesis experiment. In *Proceedings of the 7th Knowledge Based Software Engineering Conference*, pp. 6–17, IEEE Computer Society Press, Washington DC, September 1992.

Jullig, R., and Pressburger, T. 1993. Applying formal software synthesis. *IEEE Software*, May: 11–22. Los Alamitos, CA: IEEE Computer Society.

Kant, E. 1993. Synthesis of mathematical-modeling software. *IEEE Software*, May: 30–41. Los Alamitos, CA: IEEE Computer Society.

Keller, R. M., and Rimon, M. 1992. A knowledge-based software development environment for scientific model-building. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference*, pp. 192–201, IEEE Computer Society Press, Washington DC, September.

Lowry, M. 1991. Software engineering in the twenty first century. In *Automating Software Design*, Chapter 24, edited by M. R. Lowry and R. D. McCartney. AAAI Press.

Royce, W. 1993. Why software costs so much. *IEEE Software*, May: 90–91. Los Alamos, CA: IEEE Computer Society.

Schonberg, E., Schwartz, J., and Sharir, M. 1981. An automatic technique for the selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3:126–143.

Setliff, D. E., Kant, E., and Cain, J. T. 1993. Practical software synthesis. *IEEE Software*, May: 6–10. Los Alamitos, CA: IEEE Computer Society.

Setliff, D. E., and Rutenbar, R. A. 1992. Knowledge representation and reasoning in a software synthesis architecture. *IEEE Transactions on Software Engineering*, 18(6):523–533.

Smith, T. E., and Setliff, D. E. 1993. Towards design phase synthesis. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, IEEE Computer Society, Chicago IL, September.

Automated Software Engineering, 1, 215–218 (1994)

Developing Domain-Oriented Design Environments—The Question is *How*, not *Why*

JIM Q. NING
Center for Strategic Technology Research (CSTaR), Andersen Consulting, 100 South Wacker Drive, Chicago, Illinois 60606

I share many of the claims and ideas expressed in the paper, "Domain-Oriented Design Environments" (DODEs) by Gerhard Fischer. Especially, I strongly agree with the following two points:

1. *Human-Centered Design Paradigm.* According to the author, this means the "empowering and augmenting of all stakeholders in design processes to create more adequate, more understandable, and more enjoyable systems." The state of commercial acceptance of the Computer-Aided Software Engineering (CASE) tools, let alone the Knowledge-Based Software Engineering (KBSE) tools, has been disappointing. A primary reason, as pointed out by Fischer, is that the existing tools and approaches fail to emphasize active user involvement and cooperation in the software design and development process.

2. *Domain Orientation.* The existing tools may contain generic knowledge concerning computing and programming domains. Some commercial CASE tools, for example, may automatically verify structural and dataflow design constraints. More advanced tools or prototypes may use programming knowledge to automate the generation of executable and/or efficient code from high level specifications (Johnson and Feather, 1990; Kant, 1993). But typically, they do not provide problem solving support specific to the application domains for which the systems are developed.

It should be pointed out, however, that the problems raised by Fischer have long been recognized in the software engineering community. There has been on-going work in the general areas of *Domain Modeling and Analysis* (Arango, 1989; Devanbu et al., 1991; Iscoe, Williams, and Arango, 1989; Prieto-Diaz, 1990), *Software Architectures* (Garlan and Shaw, 1993), *Application Generators* (Neighbors, 1984; Batory, 1988), *Frameworks* (Johnson and Russo, 1991), *Megaprogramming* (Beohm and Scherlis, 1992; Wiederhold, Wegner, and Ceri, 1992), *Domain Specific Environments* (Griss, 1993; Ning, Miriyala, and Kozaczynski, 1994), etc. The ARPA Domain-Specific Software Architecture (DSSA) program (Mettala and Graham, 1992; Proceedings, 1990), in particular, exemplifies a coordinated effort towards developing domain-specific software tools. Despite the active research, the DODEs developed so far have not been able to scale up to demonstrate practical utility.

A large portion of Fischer's paper was contributed to argue *why* we should develop DODEs, which I do not consider to be a significant question any more. The real question

today is *how* to develop. There are many constraints and difficulties involved in developing DODEs, including:

1. *Knowledge Engineering.* Knowledge engineering or knowledge acquisition has long been recognized as a bottleneck in the application of AI and knowledge-based techniques. An effective DODE would require a large amount and a wide variety of domain-specific knowledge possibly including architectures, design frameworks, interface and interconnection standards, principles, constraints, heuristics, critics, reusable components, etc. Where does this knowledge come from? Obviously, the DODE developers cannot do it alone who typically do not possess sufficient domain knowledge. They will have to cooperate with the target users of the DODEs to conduct domain analysis. Domain analysis is known to be a hard, tedious, and time-consuming activity. What are the incentives that could potentially justify the high investment involved in the DODE knowledge engineering activities long before the developers could benefit from selling them and the users could benefit from using them?

2. *Domain Maturity.* Even with high commitment, it is difficult to imagine that DODEs can be built for those domains in which we have not developed many application systems and thus do not have good understanding. DODEs encapsulate domain-specific application development experience/knowledge accumulated in the past. The form and content of this development knowledge will keep evolving for a long period of time as a large number of applications are developed in a particular domain. A good DODE must be based on a relatively stable understanding of a domain. It should also be pointed out that even for *mature domains*, it would still be very hard to construct DODEs because of the lack of sound domain theories and systematic methods for domain analysis and modeling.

3. *Domain Specificity.* So far, DODEs have only been successfully developed for a limited number of "low-level", generic domains, such as graphical user interface (GUI) building and database generation. It may not even make sense attempting to construct DODEs for arbitrary domains. DODEs are expensive to construct. It would not justify the effort for *narrow domains* in which very few applications will ever need to be built. Besides, certain domains may not fit the cooperative, user-oriented, graphics-based design style suggested by Fischer's paper. For example, it would not be intuitive, if not totally impossible, to present domain problems visually that involve mainly complex computation but have little to do with interfaces and interconnections. For such problems, a formal, automatic programming approach could be superior.

It does not seem to me that Gerhard Fischer's paper provides a satisfactory answer to the how question, i.e., how to construct DODEs given the above difficulties. Section 3 of the paper, "A Theoretical and Conceptual Framework," only lists some guidelines or principles of what a DODE should look like. Section 5, "Domain-Oriented Design Environments (DODEs)," is a main section and was obviously intended to exemplify the basic ideas of the paper. This section provides two examples. The first one (FRAMER) supports the user interface design domain. But this domain is relatively mature and well-known. Many commercialized tools (GUI builders) are already widely used today. It is not necessary any more to justify the utility of DODEs in this particular domain. On the other hand, it is not sufficient using this example alone to show that a DODE approach would be equally effective in other, more business-oriented domains.

The second example (JANUS) is about kitchen design. This example is not only over simplifying but also misleading because it does not address software design. DODEs are supposed to be environments for designing and generating software systems, which in turn will be used to solve domain-specific problems. It is totally irrelevant whether the domain problems are design problems (e.g., the kitchen design problem) or any other problems. JANUS is just a particular software system for developing kitchen designs. A more convincing example should be a DODE for room configuration domain applications, from which a kitchen design tool such as JANUS can be generated. Otherwise, what are called Computer-Aided Design (CAD) tools that support, for a few examples, hardware board/chip design, mechanical component design, scheduling, etc., would all be classified as DODEs.

To establish some relevance of this kitchen design example with the main topic of the paper, I have to assume that the author was using it as an analogy to illustrate the ideas of what a real DODE should look like. But this analogy is weak because software systems can be fundamentally different from hardware/ physical systems. For example, it may not be as meaningful to show software components, which do not have any physical "look" or "shape" (except possibly GUI-type components), graphically. In addition, interfaces of software systems are typically loosely defined. Plugging two systems together is far more complex than, for example, putting a refrigerator next to a stove. Furthermore, many physical principles (e.g., no refrigerators on the ceiling) do not apply to software components. A simple adaptation of design frameworks successfully used in other engineering domains may not work in software engineering.

I also find some conflicting arguments in the paper. On one hand, this paper indicates that the described DODE work is complementary to the approaches pursued by KBSA research (Green et al., 1983). This is quite understandable because the existing KBSA work has mainly focused on *downstream activities* along the software lifecycle, as pointed out by Fischer. The DODE work obviously covers more *upstream activities*. If this is the case, then it is unfair to blame the KBSA work later on in Fischer's paper for its lack of emphasis on human involvement and domain orientation, which are by nature characteristics of upstream activities. The author also failed to point out that the KBSA community has been making conscious efforts recently to address issues related to collaboration support and domain orientation. Its Advanced Development Model (ADM, Andersen Consulting, 1992) project is a good example.

In general, I found the DODE paper very interesting to read. It identified some fundamental problems with the existing software engineering research and argued why a more human-centered and domain-oriented approach would be desirable. But the paper came up short explaining how the DODEs should be constructed.

References

Andersen Consulting. 1992. *Knowledge-Based Software Assistant Advanced Development Model.* Technical Proposal, July.

Arango, G. 1989. *Domain Analysis: From Art to Engineering Discipline, Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, May.
Beohm, B., and Scherlis, B. 1992. *Megaprogramming, Proceedings of the DARPA Software Technology Conference.*
Batory, D. 1988. *Concepts for a DBMS Synthesizer, Proceedings of ACM Principles of Database Systems Conference.*
Devanbu, P., Brachman, R., Selfridge, P. G., and Ballard, B. W. 1991. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 39(5).
Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C. 1983. *Report on a Knowledge-Based Software Assistant.* Technical Report RADC-TR-83-195, Rome Air Development Center, August.
Griss, M. L. 1993. Software reuse: From library to factory. *IBM Systems Journal*, 32(4).
Garlan, D., and Shaw, M. 1993. *An Introduction to Software Architectures, Advances in Software Engineering and Knowledge Engineering*, Vol. 1. World Scientific Publishing Company.
Griss, M. L., and Wentzel, K. D. 1994. *Hybrid Domain-Specific Kits for a Flexible Software Factory, Proceedings of the SAC'94*, Phoenix, AZ, March.
Iscoe, N., Williams, G., and Arango, G., (eds.) 1989. *Domain Modeling for Software Engineering.* Austin, TX: Domain-Modeling Workshop.
Johnson, W. L., and Feather, M. S. 1990. *Building an Evolution Transformation Library, Proceedings of the 12th International Conference on Software Engineering*, March.
Johnson, R. E., and Russo, V. F. 1991. *Reusing Object-Oriented Designs.* Technical Report UIUCDCS 91-1696, University of Illinois at Urbana-Champaign, May.
Kant, E. 1993. Synthesis of mathematical modeling software. *IEEE Software*, May.
Mettala, E., and Graham, M. H. (eds.) 1992. *The Domain-Specific Software Architecture Program.* Technical Report CMU/SEI-92-SR-9, Carnegie Mellon University, June.
Neighbors, J. M. 1984. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, September.
Ning, J. Q., Miriyala, K., and Kozaczynski, W. 1994. An architecture-driven, business-specific, and component-based approach to software engineering. *International Conference on Software Reuse*, submitted.
Prieto-Diaz, R. 1990. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2).
Proceedings of the Workshop on Domain-Specific Software Architectures, Hidden Valley, PA, July 1990.
Wiederhold, G., Wegner, P., and Ceri, S. 1992. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99.

Automated Software Engineering, 1, 219–222 (1994)

Commentary on "Domain-Oriented Design Environments" by Gerhard Fischer

PETER G. SELFRIDGE
AT&T Bell Laboratories, Room 2B-425, Murray Hill, NJ 07974

Introduction

"Domain-Oriented Design Environments", by Gerhard Fischer, makes a number of insightful points about the process of software design, advocates a particular style of research described as embedding "human-computer cooperative problem-solving tools into domain-oriented, knowledge-based design environments", and contrasts this approach with that emphasized by the Rome Laboratory Knowledge-Based Software Assistant (KBSA) program. The most important distinction is the focus on the "upstream" activities of problem *understanding*, as opposed to problem *solving*. This distinction leads relatively naturally to an approach of providing the "problem understanders" with a domain-oriented tool or set of tools, where underlying constraints, interactions, and domain knowledge can be represented explicitly. It also leads naturally to an emphasis on supporting the understanding process among a group of individuals, in the CSCW sense.

The points made in this paper are illustrated with two examples drawn from the large amount of excellent work coming from Fischer's group at the University of Colorado at Boulder. The first system, FRAMER, is a domain-oriented design environment (DODE) for user interface design. This general domain, the subject of numerous reports on GUI's, UIMS's, etc., is clearly a natural one for Fischer's general approach. The components of the underlying domain, graphical user interfaces, can be directly represented in a computer-based tool and a variety of supporting aids can be investigated. Fischer has examined the notion of a domain-oriented "tool kit", what he calls a "construction kit" which adds the idea of a graphical workspace, and finally, a "design environment" where additional design knowledge is provided through critics, specification sheets, and checklists. The second example system is JANUS, a DODE for kitchen design. Again, this domain is a good fit for Fischer's ideas because it maps so well onto a 2D display and the design components interact with each other primarily spatially. Still, the domain is rich in constraints that are derived from how the components will be used in the real-world, and JANUS provides various techniques for identifying and resolving design issues.

As a researcher at AT&T Bell Laboratories, I have been involved with several collaborations with a very large (> 2000 people) software organization, and my remarks are derived from this experience. In general, I am extremely sympathetic with Fischer's general approach, and agree with many of his specifics. However, I offer the following observations, and discuss the implications of these observations for Fischer's remarks and possible future areas of research.

Four Observations

Observation #1: Software Developers Spend Little Time Understanding Requirements or Writing Code. This observation has a number of implications for supporting software development. First of all, the need for improved coding tools and environments, while real, is just not a bottleneck in the organization I'm familiar with. Second, one has to ask how software developers *do* spend their time. In this organization software design takes place after a specification document is approved. This document describes in great detail the operation of a telecommunications feature from a customer perspective. Understanding this document is usually straightforward, although perhaps tedious. The software developer then begins to construct a design, embodied in a design document, which may undergo an interactive process of review and refinement. After the design document is approved, the coding process begins and is usually relatively straightforward.

So, what can we say about the actual design process? What is hard about it? In our experience, effective software design involves becoming aware of the appropriate body of "folklore" knowledge—informal knowledge in the organization about switching hardware, real-time constraints, local programming conventions, different people's areas of expertise, etc. Typically, the software designer spends a great deal of time becoming aware of this information and the most effective developers are those with the most extensive network of information sources, i.e., other people. Our work on the Designer Assistant (Selfridge, Terveen, and Long, 1992; Terveen, Selfridge, and Long, 1993, in press) has emphasized capturing and maintaining this kind of information in a computer-based tool and integrating this tool with the organizational process.

Observation #2: Software Design Doesn't Match the Visual Metaphor Very Well. While one can imagine a DODE for software design of telecommunications features, it is not clear what such a DODE would look like and exactly what benefit it might provide. Again, in my experience, designing telecommunications software is not particularly visual. Some visual notations are used, such as finite-state diagrams to represent message handling and other state-based computation, and tools are used to validate such representations. However, the majority of design work doesn't seem to require a graphical workspace approach.

There are two responses to this line of thinking. First, it could be argued that if we understood the notion of a telecommunications feature and were more rigorous with formally describing such (Zave, 1993), the idea of a DODE for designing telecommunications features would be much natural. I have little to say about this possibility beyond its plausibility. Second, it could be that a DODE for this purpose could provide more generic CSCW benefit by facilitating issue generation and resolution among groups of designers. This could be a very valuable line of inquiry; however, it doesn't really match the idea of a DODE as described by Fischer.

Observation #3: Legacy Systems Dominate Large Software Design and Impose Special Challenges. In the organization I'm familiar with, design activity is dominated by two things. The first is folklore knowledge, discussed above, and the second is the current software system. The design activity is essentially to modify and add to the existing software structure to implement the requirements while not breaking anything else. In addition, the modifications should be as parsimonious as possible (however, time pressures usually negate this goal).

The existence of very large legacy systems in the form of code (other information is rarely, if ever, preserved) is a serious challenge to the DODE concept for this area. How exactly might information in these large systems get acquired and integrated into a design environment? While various research in reverse engineering (Waters and Chikofsky, 1993) (including our own work (Devanbu et al., 1991; Selfridge, 1991) has illustrated the ability to derive some information from old code, this information rarely captures meaningful semantics.

Observation #4: Organizational Process Maturity is Absolutely Critical to Providing Effective Computer Support. This is an observation that I think Gerhard Fischer would agree with completely. In our experience, understanding of organizational process and ownership and measurement of such processes are critical in improving them with or without computer support. This has been observed by others as well (Royce, 1992). In our work on the Designer Assistant, integrating the tool with t, integrating the tool with current practice as prescribed by a process description was important in three ways. First, it allowed us to understand where and how such a tool could be useful. Second, it coerced, in an acceptable way, users into actually using the tool at certain appropriate times, maximizing the possibility that it would be useful. Finally, it allowed us to address the knowledge maintenance problem is a disciplined way, by capturing disagreements with and desired additions to the knowledge.

However, the issue of process maturity and technology transfer does raise serious issues on how research into software design should take place. One extreme view is that academic research in this area is destined to be irrelevant, since such research is not embedded in a real "customer" organization. Of course, there is a spectrum of degrees to which academic research can be coupled to real-world concerns, and Fischer's work often is done in various sorts of collaboration with outside companies, and he always takes his inspiration from the real world. He is also an outspoken advocate of empirical testing and user experimentation. However, how effective his approach will be when tested in a real software development organization is an open question.

Conclusions

Gerhard's vision of domain-oriented design environments, initially "seeded" by a special startup effort but from then on "living and growing" through interaction in a work environment is a compelling one. His positioning of this idea as complementary to more traditional approaches like KBSA (which, by the way, are beginning to embrace many of the ideas he espouses) is correct. Furthermore, I agree with this vision and with much of his work in supporting human-centered design. However, as he himself admits, it remains to be tested in the real world and such tests will force the vision, the technology, and the way people work to evolve in mutually supportive way.

References

Devanbu, P., Brachman, R. J., Selfridge, P. G., and Ballard, B. W. 1991. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34:34–49.

Royce, W. 1992. Why industry often says "no thanks" to research. *IEEE Software*, November: 97–99.

Selfridge, P. G. 1991. Knowledge representation support for a software information system. In *Proceedings of the 7th IEEE Conference on AI Applications*, pp. 134–140, Miami Beach, Florida.
Selfridge, P. G., Terveen, L. G., and Long, M. D. 1992. Managing design knowledge to provide assistance to large-scale software development. In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference (KBSE-92)*, pp. 163–170, Tyson's Corner, Virginia, September 22–25. Available from the IEEE Press.
Terveen, L. G., Selfridge, P. G., and Long, M. D. 1993. From "Folklore" to "Living Design Memory". In *Proceedings of INTERCHI-93*, pp. 15–22, Amsterdam, The Netherlands, April 24–29. Available from ACM.
Terveen, L. G., Selfridge, P. G., and Long, M. D. (in press) *Living Design Memory for Software Development: Framework, System, Lessons Learned.*
Waters, R. C., and Chikofsky, E. J. (eds.) 1993. *Working Conference on Reverse Engineering, Baltimore, Maryland, May 21–23.* Available from the IEEE Computer Society.
Zave, P. 1993. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26:20–31.

Automated Software Engineering, 1, 223–229 (1994)

Domain-Oriented Design Environments: Reply to Commentaries

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

A. Sutcliffe, J. Ning, P. Selfridge and D. Setliff have commented on my article "Domain-Oriented Design Environments" and I would like to thank them for their insightful comments. One should not be surprised that researchers hold different views about an area as complex and as volatile as Software Engineering, especially because my paper not only describes some past achievements, but also outlines a research agenda for the future. I would like to thank the editors of *Automated Software Engineering* who have granted me the privilege of replying to the comments by A. Sutcliffe, J. Ning, P. Selfridge, and D. Setliff. I have organized my reply around themes, using the names of these individuals as references. I have chosen often to use "we" instead of "I" to acknowledge the group of collaborators at CU Boulder and elsewhere, who share with me the same view.

Design

Design is concerned with "how things ought to be in order to attain goals, and to function" (Simon, 1981). Design understood this way is more than "the act of translating requirements into specifications and constraints" (Setliff). Design complements the natural sciences, whose primary goal is to analyze. Designers not only solve given problems by reasoning about formal representations, but they (architects, industrial designers, curriculum designers, or software designers) have to get actively involved in framing problems. Designers are not the sole owners of problems. They have to collaborate with all stakeholders (clients, customers, other designers) in a mutual education process to understand problems and construct the knowledge for solving them. Design methods will be deeply influenced by the artifacts developed. The design of computational artifacts to empower humans faces different issues than the design of technical systems, such as VLSI CAD design (Setliff) or compilers.

Problems of Domain-Oriented Design Environments

What is a Domain?

Sutcliffe raises the issue that "there is no sound theory about what a 'domain' is." I agree that domains cannot be precisely defined—they are part of the design activity themselves

(so they change when goals change). We try to define domains in our environments (such as departments in universities, or professional societies), and they serve as useful constructs. But at the same time, we call for interdisciplinary research to acknowledge that real world problems do not fit into our preconceived domains. Domains and their boundaries will undergo change as our world changes. This is specifically acknowledged in our work by postulating our model of seeds, evolutionary growth and reseeding.

I disagree with the assertion that "it is difficult to imagine that DODEs can be built for immature domains" (Ning). Our research has demonstrated that it can be a very fruitful endeavor to create DODEs for immature domains (and we have done so for lunar habitat design, for computer networks, etc.). By creating DODEs through intensive collaboration with domain experts, we have shown that these efforts can make major contributions toward deepening our understanding of a domain.

What is the Price of Working in a Domain?

Sutcliffe asserts that "DODE's domain-specific nature will limit application to a small set of related problems, leaving only an outline architecture as a more general result." This is an adequate characterization and it is supported by the results of our work. We are aware of the tension and the design trade-off between the Turing Tar-Pit (as articulated by Alan Perlis) "The Turing Tar Pit: everything is possible but nothing of interest is easy" and the inverse of it "The over-specialized system: everything is easy, but nothing of interest is possible." Referring back to human organizations and domain expertise again: our society educates its members in domains, and switching from one domain to another is a non-trivial undertaking. So why should we expect that we will get DODEs for free? There is growing wide spread recognition and a growing number of computational artifacts that demonstrate that domain orientation will allow us to develop new generations of human-centered computational artifacts (e.g., Mathematica for mathematicians, spreadsheets for planning and decision making, drawing and painting software for artists, etc.) by supporting *human problem-domain* communication with the goal of narrowing the gap between subject domain and computational substrate.

We are working on substrates and layered architectures to increase the sharing of components between DODEs in related domains. But without paying the price of working in a domain, our computational environments will be severely limited in the amount (1) of support they can provide (e.g., there would be no work-triangle critic without domain knowledge), and (2) of end-user control and interest (e.g., end-users are not interested in the computer per se, but in their tasks).

Knowledge Acquisition

Ning observes that "an effective DODE will require a large amount and a variety of domain-oriented knowledge." Our process model, based on seeds, evolutionary growth, and reseeding (Fischer et al., 1994), is an important alternative to the conventional approaches of

knowledge acquisition as well as the futuristic approaches of machine learning pursued in AI-oriented research efforts. Our model explicitly acknowledges the fact that (1) human knowledge is tacit (Polanyi, 1966) (so the best we can hope for is a seed), (2) knowledge changes over time (requiring support for evolutionary growth), (3) the breakdowns based on lack of knowledge will be experienced by the domain designers and not by the environment developer (making end-user modifiability a necessity rather than a luxury), and (4) social incentives and rewards for providing and documenting this knowledge (e.g., in the form of design rationale) may be more important than the particular formalism chosen for its representation.

How, Not Why

Ning states "that the real question today is how to develop, rather than why we should or should not develop DODEs." Our research prototypes (see references in my paper) demonstrate that we have some understanding of "how" one goes about building DODEs. Beyond that, we assisted others in developing DODEs and demonstrated the practical value of some of our DODEs in industrial research environments (e.g., the voice dialog design environment in use at USWest Advanced Technologies (Repenning and Sumner, 1992), the service-provisioning environment in use at NYNEX (Ostwald, Burns, and Morch, 1992), and the lunar habitat environment in use by a NASA contractor (Stahl, 1993)).

An important aspect of DODEs is the possibility to construct them *incrementally* (e.g., the voice dialog design environment existed and was used by domain workers for more than a year before a critiquing component was added), and to emphasize different components for different domains (e.g., the simulation component is of great importance in the voice dialog design environment).

Scaling Up

Scaling up is a critical issue for DODEs as it is for any other computational environment. Our work so far demonstrated (1) that the "seeds—evolutionary growth—reseeding" model provides a good foundation for scaling up, and (2) that many of the integration components assist users in dealing with information spaces that are too large to be explored by browsing only. DODEs acquire a partial understanding of the task at hand by analyzing the partial construction and the partial specification. The CONSTRUCTION-ANALYZER and CATALOG-EXPLORER exploit this partial understanding to locate relevant argumentation and catalog examples for the user. Following Sutcliffe's observation that for large information spaces "intelligent retrieval engines will be necessary," we have explored such mechanisms for several years (Fischer, Henninger, and Redmiles, 1991) and incorporated them in our DODEs (Nakakoji, 1993).

The Proper Role of Automation

Understanding the Proper Role of Humans and Computers in Joint Human-Computer Systems

Even strong advocates of automated systems such as expert systems' researchers acknowledge that "most knowledge-based systems are intended to be of assistance to human endeavor; they are almost never intended to be automatic agents. A human-machine interaction subsystem is therefore a necessity" (Feigenbaum and McCorduck, 1983). The proper role of humans and computers has been explored in numerous areas (to name just a few examples: in machine translation (Kay, 1980), in cockpit design (Billings, 1991), and in the general foundations for tool and system design (Illich, 1973; Fischer, 1990)). The question of the proper role of automation is raised succinctly by Billings (1991): "During the 1970's and early 1980's ... the concept of automating as much as possible was considered appropriate. The expected benefits were a reduction in pilot workload and increased safety. Although many of these benefits have been realized, serious questions have arisen and incidents/accidents have occurred which question the underlying assumption that the maximum available automation is always appropriate or that we understand how to design automated systems so that they are fully compatible with the capabilities and limitations of the humans in the system" (p. 4). Contrary to Sutcliffe's claim that "every domain will have to have an exhaustive analysis to find all the principles, rules, guidelines etc., for good design," critiquing components embedded in DODEs do not require any kind of completeness. While it is highly desirable that a substantial amount of critiquing knowledge gets accumulated over time, a system with a just a few critiquing rules can greatly increase the usability of a DODE.

Lack of any Theoretical Basis for Cooperation

Sutcliffe observes that "unless design of software tools is based on a sound analysis of how the user and machine cooperate to achieve designs we run the risk of providing inappropriate functionality which may either over-automate or under-support the designer's job." Understanding cooperation is a critical challenge not only for KBSAs and DODEs, but for all intellectual teamwork (Galegher, Kraut, and Egido, 1990). Our work on DODEs should not and cannot wait until the theoretical basis for cooperation will exist, but we attempt with our efforts to contribute to the creation of this basis. Our work is guided by principles for collaboration, such as (1) all stakeholders must be involved (to account for the "symmetry of ignorance" (Rittel, 1984)), (2) to be involved, the stakeholders must be informed in an understandable way (requiring that representations are developed that can serve as "languages of doing" (Ehn, 1988)), and (3) there must be shared knowledge (including knowledge of each other's intent (Resnick, Levine, and Teasley, 1991)).

Integrating KBSAs and DODEs

In Setliff's view, "current software synthesis architectures are well on their way toward
stantiating Fischer's DODE design process"—indicating that many recent research eff
emerging from the instantiation of the original KBSA effort moved toward some of
goals of DODEs. I see a natural symbiosis between the two research directions: KB
emphasize downstream activities and DODEs emphasize upstream activities. This vie
shared by Selfridge when he observes about our work: "The most important distinctic
the focus on the 'upstream' activities of problem *understanding*, as opposed to prob
solving." Obviously, either approach cannot ignore the other phase (e.g., we have built
eral computational substrates serving as lower layers in DODEs (Repenning and Sum
1992), and the KBSA efforts have pursued upstream activities in the context of requ
ments engineering (Proceedings, 1993). But the different emphasis has led to a nun
of differences: KBSAs and DODEs investigated different classes of problems, looked
different disciplines for help and ideas, and approached the human role and assessn
studies from different angles.

Problems are Different

Sutcliffe observes that "for safety critical domains, formal approaches and automatic p
gramming is not only desirable but essential. However, Fischer reminds us that m
problems do not fall into this class." There is no doubt that we need correct and effic
programs (just as we need buildings that do not collapse), but what is the value of th
programs if they are not relevant, suitable, adequate, or enjoyable to users in their
situation (just as houses are judged by more criteria than that they do not fall down
also claim that the scientific community needs a better understanding of the limitation
formal methods in safety critical systems (e.g., the accident in the Persian Gulf in wh
an airliner relying on the AEGIS system was shot down represents a design disaster
formal methods would not have prevented (Lee, 1992)).

Where Do We Look for Ideas and Help?

Historically, computer science has looked to mathematics and logic to create a foun
tion (and these disciplines served well for improving "downstream" activities). But a
the foundations have been established, other disciplines may be more important, such
cognitive psychology (to better understand the human part), social sciences (to underst
collaboration), evolution (to understand the nature of complex systems), and architect
(to understand design as an activity that needs to define and create contexts and not o
operate in given contexts). In the long run, I think that "Software Engineering" may
the wrong term because it focuses on the medium rather than on the characterizatior
domains (in mature design domains, we do not speak of "steel" or "concrete" engineeri
but of "civil" or "electrical" engineering).

Do Not Postulate a New Human

Simon (1981) acclaims the framers of the U.S. Constitution by noting that "they did not postulate a new man to be produced by the new institutions but accepted as one of their design constraints the psychological characteristics of men and women as they knew them, their selfishness as well as their common sense" (p. 163). It may be that what is wrong with the logical and mathematical design methods is that they are the product of a mode of reasoning alien to design (Rittel, 1984). A human-centered view toward design should take into account that "logic is most definitely not a good model of human cognition. Humans take into account both the content and the context of the problem, whereas the strength of logic and formal symbolic representations is that the content and context are irrelevant. Taking content into account means interpreting the problem in concrete terms, mapping it back onto the known world of real actions and interactions" (Norman, 1993) (p. 228). This and other observations such as (1) humans enjoy doing and deciding, (2) humans act until breakdowns occur, (3) humans operate by using information in the world as an important resource, and (4) domain-orientation preserves content and context, have served as guiding principles for our work on DODEs to complement the more formal approaches pursued in the KBSA communities.

References

Billings, C. E. 1991. *Human-Centered Aircraft Automation: A Concept and Guidelines*. NASA Technical Memorandum 103885, NASA Ames Research Center, Moffett Field, CA, August.

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

Feigenbaum, E. A., and McCorduck, P. 1983. *The Fifth Generation. Artificial Intelligence and Japan's Computer Challenge to the World*. Reading, MA: Addison-Wesley Publishing Company.

Fischer, G. 1990. Communications requirements for cooperative problem solving systems. *The International Journal of Information Systems* (Special Issue on Knowledge Engineering), 15(1):21–36.

Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. (in press). Seeding, evolutionary growth and reseeding: Supporting incremental development of design environments. In *Human Factors in Computing Systems, CHI'94 Conference Proceedings (Boston, MA)*.

Fischer, G., Henninger, S. R., and Redmiles, D. F. 1991. Cognitive tools for locating and comprehending software objects for reuse. In *Thirteenth International Conference on Software Engineering (Austin, TX)*, IEEE Computer Society Press, ACM, IEEE, Los Alamitos, CA, pp. 318–328.

Galegher, P., Kraut, R., and Egido, C. (eds.) 1990. *Intellectual Teamwork*. Hillsdale, NJ: Lawrence Erlbaum Associates.

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

Kay, M. 1980. *The Proper Place of Men and Machines in Language Translation*. Technical Report CSL-80-11, Xerox Palo Alto Research Center, October.

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

Nakakoji, K. 1993. *Increasing Shared Understanding of a Design Task Between Designers and Design Environments: The Role of a Specification Component*. Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado. Also available as Technical Report CU-CS-651-93.

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

Ostwald, J., Burns, B., and Morch, A. 1992. *The Evolving Artifact Approach to System Building, Working Notes of the AAAI 1992 Workshop on Design Rationale Capture and Use*, AAAI, San Jose, CA, July, pp. 207–214.

Polanyi, M. 1966. *The Tacit Dimension*. Garden City, NY: Doubleday.

Proceedings of IEEE International Symposium on Requirements Engineering. 1993. IEEE Computer Society, IEEE Computer Society Press, Los Alamitos, CA, January.

Repenning, A., and Sumner, T. 1992. Using Agentsheets to create a voice dialog design environment. *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, ACM Press, pp. 1199–1207.

Resnick, L. B., Levine, J. M., and Teasley, S. D. (eds.) 1991. *Perspectives on Socially Shared Cogni* Washington, DC: American Psychological Association.

Rittel, H. W. J. 1984. Second-generation design methods. In *Developments in Design Methodology*, edite N. Cross, pp. 317–327. New York: John Wiley & Sons.

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

Stahl, G. 1993. *Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Sup of Cooperative Design*. Ph.D. Dissertation, Department of Computer Science, University of Colorado, Boul CO.