Complex Systems: Why Do They Need to Evolve and How Can Evolution Be Supported

Gerhard Fischer

University of Colorado, Center for LifeLong Learning & Design (L^3D)
Department of Computer Science, Campus Box 430
Boulder, CO 80309-0430, USA
email: gerhard@cs.colorado.edu

Abstract. We live in a world characterized by evolution—that is, by ongoing processes of development, formation, and growth in both natural and human-created systems. Biology tells us that complex, natural systems are not created all at once but must instead evolve over time. We are becoming increasingly aware that evolutionary processes are ubiquitous and critical for social, educational, and technological innovations as well.

The driving forces behind the evolution of these systems is their use by communities of practice in solving real-world problems as well as the changing nature of the world, specifically as it relates to technology. The seeding, evolutionary growth, and reseeding model is a process description of how this happens. By integrating working and learning in communities of practice, we have created organizational memories that include mechanisms to capture and represent task specifications, work artifacts, and group communications. These memories facilitate organizational learning by supporting the evolution, reorganization, and sustainability of information repositories and by providing mechanisms for access to and delivery of knowledge relevant to current tasks.

Our research focuses specifically on the following claims about design environments embedded within dynamic human organizations: (1) they must evolve because they cannot be completely designed prior to use; (2) they must evolve to some extent at the hands of the users; (3) they must be designed for evolution; and (4) to support this approach with World-Wide Web technology, the Web has to be more than a broadcast medium; it has to support collaborative design.

Keywords: design; evolution; domain-oriented design environments; seeding, evolutionary growth, and reseeding model (SER model); open versus closed systems

Acknowledgments. The author would like to thank members of the Center for LifeLong Learning & Design at the University of Colorado who have made major contributions to the conceptual framework and systems described in this paper. Jim Ambach and Ernesto Arias contributed to the conceptual framework and scenario development. The research was supported by (1) the National Science Foundation, Grants REC-9631396 and IRI-9711951; (2) NYNEX Science and Technology Center, White Plains; (3) Software Research Associates, Tokyo, Japan; (4) PFU, Tokyo, Japan; and (5) Daimler-Benz Research, Ulm, Germany.

1 Introduction

1.1 Necessity for Evolution

The basic assumption that complete and correct requirements can be obtained at some point of time is theoretically and empirically wrong. 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 [23]. A consequence of the "thin spread of application knowledge" [10] 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.

Design methodologists (e.g., [43,44]) demonstrate with their work that the design of complex systems requires the integration of problem framing and problem solving and 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 and implemented. The conceptual structures underlying complex software systems are too complicated to be specified accurately in advance and too complex to be built faultlessly [7]. Specification and implementation have to co-evolve [47] requiring the owners of the problems [15] to be present in the development. While evolution is no panacea and creates its own problems, there are strong reasons to increase the efforts and the costs to include mechanisms for evolution (such as end-user modifiability, tailorability, adaptability, design rationale, making software “soft”) into the original design of complex systems. Experience has shown [8,25] that the costs saved in the initial development of system by ignoring evolution will be spent several times over during the use of a system.

1.2 Theory about Evolution of Complex Systems

Evolution of complex systems is a ubiquitous phenomenon. Many researchers have addressed the evolutionary character of successful complex systems and of scientific endeavor. Laszlo [28] stresses that a new paradigm is emerging in many fields, leading to a replacement of earlier ideas that were based on mechanistic determinism toward new models of change, indeterminance and evolution. Popper [36] reminds us that knowledge should be open to critical examination and that the advance of knowledge consists in the modification of earlier knowledge. Dawkins [11] demonstrates that big-step reductionism cannot work as an explanation of mechanism; we can't explain a complex thing as originating in a single step, but complex things *evolve* (implying that models from biology may be more relevant to future software systems than models from mathematics).

1.3 The Evolution of Complex Software Systems

Evolution is especially essential in software systems. The assumption of complete requirements at any point in time is detrimental to the development of successful (i.e., useful and usable) software systems. Brooks [7] argues that *successful* software gets changed because it offers the possibility to evolve. Lee [29] describes many convincing examples (including the failure of the Aegis system in the Persian Gulf) that design approaches based on the assumption of complete and correct requirements do not correspond to the realities of this world. Curtis and colleagues [10] identify in a large-scale empirical investigation that fluctuating and conflicting requirements are critical bottlenecks in software production and quality. The Computer Science and Technology Board [8] provides empirical data that 40-60 percent of the lifecycle costs of a complex system is absorbed by maintenance and 75 percent of the total maintenance efforts are enhancements. Much of this cost is due to the fact that a considerable amount of essential information (such as design rationale [18,31]) is lost during development and must be reconstructed by the designers who maintain and evolve the system. In light of these data, development and maintenance have to merge into cycles of an evolutionary process making capturing of design rationale a necessity rather than a luxury.

1.4 Claims about Evolution

Software design needs to be understood as an evolutionary process where system requirements and functionality are determined through an iterative process of collaboration among multiple stakeholders (including developers and users). Requirements cannot be completely specified before system development occurs. Our previous study of design processes support the following claims:

- *Software systems must evolve; they cannot be completely designed prior to use*. Design is a process that intertwines problem solving and problem framing [43]. Software users and designers will not fully determine a system's desired functionality until that system is put to use. Process models that describe the different phases of the software life cycle need to take advantage of this fact.
- *Software systems must evolve at the hands of the users*. End users experience a system's deficiencies; subsequently, they have to play an important role in driving its evolution. Software systems need to contain mechanisms that allow end-user modification of system functionality.
- *Software systems must be designed for evolution*. Through our previous research in software design, we have discovered that systems need to be designed *a priori* for evolution. Software architectures need to be developed for software that is designed to evolve.

2 Domain-Oriented Design Environments (DODEs)

Domain-oriented design environments (DODEs) [14] are software systems that support design activities within a particular domain such as the design of kitchens, voice dialog systems, and computer networks. DODEs are a particularly good example of complex software systems that need to evolve. Design within a particular domain typically involves several stakeholders whose knowledge can be elicited only within the context of a particular design problem. Different stakeholders include the developers of a DODE (environment developers), the end users of a DODE (domain designers), and the people for whom the design is being created (clients). To effectively support design activities, DODEs need to increase communication between the different stakeholders and anticipate and encourage evolution at the hands of domain designers.

Software systems (such as DODEs) model parts of our world (e.g., the physical computer networks consisting of computers, networks, etc.). Our world evolves in numerous dimensions as new artifacts appear, new knowledge is discovered, and new ways of doing business are developed. There are fundamental reasons why systems cannot be done "right" at the beginning. Successful software systems need to evolve.

2.1 Design Environments: Limited Scope, Better Support

Not *all* problems in the development of *any* complex software can be solved by one (and always the same) approach. The more specifically we address a certain kind of complex software, the more likely will we be able to find effective support for its evolutionary development. Henderson and Kyng [25] demonstrate that enhancements extending through the lifetime of a complex system are critical. Norman [34] shows that design and evolution have many things in common. Simon [45] provides convincing evidence that complex systems evolve faster if they can build on stable subsystems.

In our work on DODEs, we build on object-oriented techniques, but transcend pure, basic object-orientation by integrating and embedding object abstractions in the specific context of *design in an evolving domain*. Like design patterns [1,21] and application frameworks [30], DODEs provide a meaningful context for evolution [13]. DODEs are even more powerful than the other above-mentioned contextualizations, as they take the domain into account.

2.2 Domain Orientation: Situated Breakdowns and Design Rationale

Domain-oriented systems are rooted in the context of use in a domain. While the DODE approach itself is generic, each of its applications is a particular domain-oriented system. Our emphasis on *domain-oriented* design environments acknowledges the importance of situated and contextualized communication and design rationale as the basis for *effective* evolutionary design. Polanyi [35] analyzes the observation that human knowledge is tacit (i.e., we know more than we can say) and that some of it will be activated only in actual problem situations. In early knowledge-based system building efforts, there was a distinct knowledge acquisition phase that was assumed to lead to complete requirements—contrary to our assumption of the seeding, evolutionary growth, reseeding (SER) model (presented later in the paper). The notion of a "seed" in the SER model emphasizes our interpretation of the initial system as a catalyst for evolution—evolution that is in turn supported by the environment itself.

2.3 End-User Modification and Programming for Communities: Evolution at the Hands of Users

Because end users experience breakdowns and insufficiencies of a design environment in their work, *they* should be able to report, react to, and resolve those problems. Mechanisms for end-user modification and programming are, therefore, a cornerstone of evolvable systems. At the core of our approach to evolutionary design lies the ability of end users (in our case, domain designers) to make significant changes to system functionality, and to share those modifications within a community of designers. It is their perception that should determine what is considered urgent to change, not the risks determined by developers. The types of changes that must occur during the evolutionary growth of a system go beyond the setting of predefined parameters or preferences and include the ability to alter system behavior in non-trivial ways. Winograd [49] argues why design environments are needed to make end-user programming feasible. We don't assume that all designers will be willing or interested in making system changes, but drawing upon the work of Nardi [33] we do know that within local communities of software use there often exist local developers and power users who are interested in and capable of performing these tasks.

3 Evolutionary Design at Work: A Scenario

The following scenario illustrates how a design environment can affect the exemplary domain of computer network design. The scenario emphasizes the importance of *end-user driven evolution.* The system described, *NetDE* (see Fig. 1 and Fig. 2), is a DODE for the domain of computer network design. NetDE incorporates several principles including:

- Domain-oriented components that provide domain designers (in this case, computer network designers) the capability to easily create design artifacts; these domain designers are the end-users of NetDE.
- Features that allow the specification of design constraints and goals so that the system understands more about particular design situations and gives guidance and suggestions for designers relevant to those situations.
- Mechanisms that support the capture of design rationale and argumentation embedded within design artifacts so that they can best serve the design task.

- Mechanisms that support end-user modifiability so that the network designers experiencing deficiencies of NetDE can drive the evolution of the system.
- Features that increase communication between the system stakeholders (i.e., designers of NetDE and the network designers using NetDE).
- The integration of communities of practice within the evolution of NetDE.

This scenario involves two network designers (D_1, D_2) at the University of Colorado who have been asked to design a new network for clients within the Publications Group in the dean's office at the College of Engineering.

3.1 Evolution of Design Artifacts: Designing a New Network

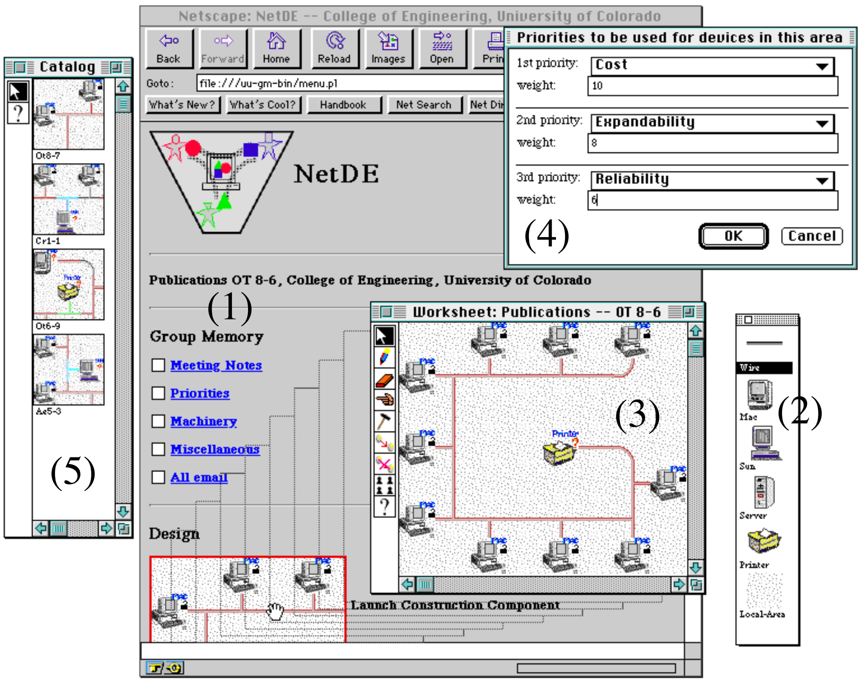
D_1's clients are interested in networking ten newly purchased Macintosh Power PCs and a laser printer. Through a combination of email discussions and meetings, D_1 learns that the clients want to be able to share the printer, swap files easily, and send each other email. D_1 raises the issue of connecting to the Internet, and was told that the clients would be interested at some point, but not for the time being. It was also made clear that the clients had spent most of their budget on the computer hardware, and did not have much left over for sophisticated network services and tools.

From our previous work in network design, we know that design specification and rationale comes from a number of stakeholders, including network designers and clients, and is captured in different media including email and notes. To be most effective, this rationale needs to be stored in a way that allows access to it from the relevant places within a design.

D_1 invokes the NetDE system. A World-Wide Web (WWW) Browser appears on the desktop presenting a drawing of the College of Engineering. Every network and subnet in the College of Engineering can be accessed by navigating through different parts of the drawing. By selecting the "New Design" option, D_1 is presented an empty NetDE page that he names "Publications OT 8-6" after the office where the clients are located. The new page becomes a repository for all of the background information and rationale that D_1 has regarding the new network. This is achieved by sending all email and text files that D_1 has to the (automatically created) email address "PublicationsOT8-6." NetDE insures that the WWW page immediately updates itself to show links to the received mails and files (Fig. 1, (1)).

Selecting the "Launch Construction Component" option opens a palette of network objects (Fig. 1, (2)) as well as a worksheet (3). D_1 starts by specifying certain design constraints to the system (4). Immediately the Catalog (5) displays a selection of existing designs that have constraints similar to those specified by D_1. Selecting one of the designs represented in the Catalog moves that design into the worksheet where D_1 can modify it. D_1 changes the design to reflect the specific needs of the Publications Group. NetDE is accessible through the World-Wide Web, so that other network designers (D_2...D_n) can use it, also. The existing designs may be contributions from other designers.

WWW access is crucial for maintaining a distributed community of practice. The Behavior Exchange [39] is addressing these needs. The capture of design rationale and argumentation can occur through the use of a group memory. The GIMMe system [19] explores the creation and maintenance of a group memory accessible through the WWW.



Netscape: NetDE -- College of Engineering, University of Colorado

Goto: file:///uu-gm-bin/menu.pl

Publications OT 8-6, College of Engineering, University of Colorado

Group Memory

- Meeting Notes
- Priorities
- Machinery
- Miscellaneous
- All email

Design

Launch Construction Component

Priorities to be used for devices in this area

1st priority: Cost
weight: 10

2nd priority: Expandability
weight: 8

3rd priority: Reliability
weight: 6

Worksheet: Publications -- OT 8-6

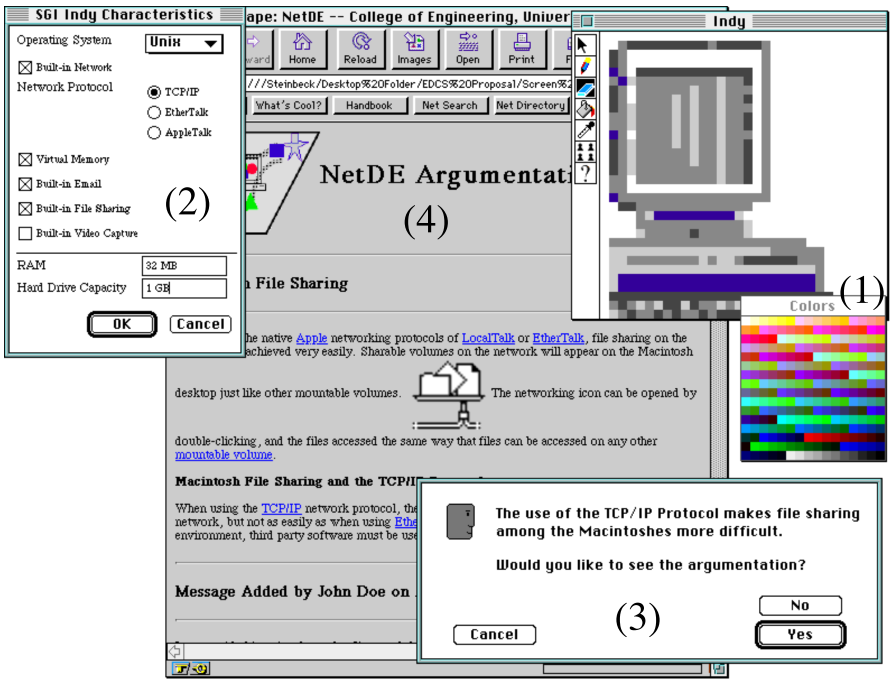
Fig. 1. NetDE in Use

Finally, NetDE provides a domain specific construction mechanism (the palette and the worksheet), and allows the specification of design constraints and goals. Using additional specification mechanisms, D_1 describes how the network will be used, and what kinds of networking services are desired. This is the first time D_1 has networked Macs, so he takes advantage of the NetDE critiquing feature, which will evaluate his design and compare it to the established design constraints. During evaluation, NetDE suggests the use of the EtherTalk network protocol, and the PowerTalk email capabilities that come standard with the Macs. D_1 agrees with this assessment because they limit the cost of the network. He finishes creating his design.

Integration of specification, construction, catalog, and argumentation components is the characteristic strength of a DODE such as NetDE. These components and their interaction are critical to the "evolvability" of the system. The process D_1 and D_2 follow (below) is an instantiation of our seeding-evolutionary growth-reseeding process model.

3.2 Evolution of a Design Environment: NetDE

Several months pass, and Publications is interested in changing its network. D_1 is not available, so D_2 is to design the new changes. D_2 receives email from Publications indicating that their network needs have changed. They want to start publishing WWW pages and will need Internet access. They will also be using a Silicon Graphics Indy computer. They have received a substantial budget increase for their network.



Operating System: Unix
☒ Built-in Network
Network Protocol: [✓ TCP/IP] EtherTalk AppleTalk
☒ Virtual Memory
☒ Built-in Email
☒ Built-in File Sharing
☐ Built-in Video Capture
RAM: 32 MB
Hard Drive Capacity: 1 GB

NetDE Argumentat

n File Sharing

he native Apple networking protocols of LocalTalk or EtherTalk, file sharing on the achieved very easily. Sharable volumes on the network will appear on the Macintosh desktop just like other mountable volumes. The networking icon can be opened by double-clicking, and the files accessed the same way that files can be accessed on any other mountable volume.

Macintosh File Sharing and the TCP/I

When using the TCP/IP network protocol, the network, but not as easily as when using Ethe environment, third party software must be use

Message Added by John Doe on

The use of the TCP/IP Protocol makes file sharing among the Macintoshes more difficult.

Would you like to see the argumentation?

Fig. 2. The Network Evolves

First, D_2 accesses the NetDE page that describes the Publications network. She quickly reviews the current design and rationale to learn what has already occurred. She updates the design specification to reflect the fact that cost is no longer as important, and that speed has become more important. Then, she searches the NetDE palette to see if it has an icon representing the Indy. She does not find one, and realizes that it must be added. After reviewing the specs for the Indy from the Silicon Graphics Web Page, D_2 creates a new palette element for the Indy (Fig. 2, (1)), and then defines its characteristics using some of NetDE's end user modifiability features (Fig. 2, (2)). According to the company's specs, the Indy has built-in networking capabilities, and understands the TCP/IP network protocol. D_2 enters this information, and the new icon appears in the palette.

Since breakdowns are experienced by end users, they need to be able to evolve the system's functionality. This calls for the development of specialized mechanisms that allow end users to alter system functionality without having to be computer programmers. In order for NetDE to take full advantage of new objects added by the network designer, it must provide facilities that define not just the look of the new object, but also its behavior.

D_2 adds the Indy to the design, and NetDE indicates (by displaying different colored wires) that the two types of machines (Macs and the Indy) are using different network protocols. D_2 knows that Macs can understand TCP/IP protocol, so she changes the network's protocol to TCP/IP. After invoking NetDE's critiquing mechanism, D_2 receives a critiquing message indicating that the use of TCP/IP violates the easy file-sharing design constraint (Fig. 2, (3)). After reading through some of the argumentation (Fig. 2, (4)), D_2 learns that although file sharing is possible in TCP/IP with the Macs, it is not as easy as when they are using EtherTalk. D_2 decides that this is not a constraint she would like to break, and decides to ask some other network designers if there is a way to get the Indy to understand EtherTalk. D_2 learns that there is software the Indy can run to translate protocols, and she adds an annotation to the Indy object to reflect this.

A critiquing component is important in linking design rationale and argumentation to the designed artifact as well as for pointing out potential breakdowns to the designer. Very drastic changes (like the introduction of wireless communication) will not be covered by the end-user modification mechanisms. In those cases, the ability to describe system changes to environment developers is critical for maintaining communication among different stakeholders of the system. When unexpected modification needs occur, users must be able to articulate their needs and notify developers. The information provided to the environment developer will be useful to describe how the system is being used and what sort of issues the system is not addressing, leading to subsequent radical evolution of the system.

4 Computer-Supported Evolutionary Design

The above scenario illustrates both (1) the DODE itself evolves and (2) how artifacts created with the DODE evolve at the hands of end-users (such as network designers). The ability of a DODE to co-evolve with the artifacts created within it makes the DODE architecture the ideal candidate for creating an evolvable application family. In the following, we describe the SER process model as a systematic way to structure evolution [20].

The domain orientation of a design environment enriches (1) the amount of support that a knowledge-based system can provide, and (2) the shared understanding among stakeholders. Design knowledge includes domain concepts, argumentation, case-based catalogs, and critiquing rules. The appeal of the DODE approach lies in its compatibility with an emerging methodology for design [9,12], views of the future as articulated by practicing software engineering experts [8], reflections about the myth of automatic programming [42], findings of empirical studies [10], and the *integration* of many recent efforts to tackle specific issues in software design (e.g., recording design rationale [18], supporting case-based reasoning [38], creating artifact memories [48], and so forth).

4.1 Seeding, Evolutionary Growth, Reseeding—The SER Process Model for DODEs

Because design in real world situations deals with complex, unique, uncertain, conflicted, and unstable situations of practice, design knowledge as embedded in DODEs will never be complete because design knowledge is tacit (i.e., competent practitioners know more than they can say) [35], and additional knowledge is triggered and activated by actual use situations leading to breakdowns [16]. Because these breakdowns are experienced by the users and not by the developers, computational mechanisms that supporting end-user modifiability are required as an intrinsic part of a DODE.

Three intertwined levels can be distinguished whose interactions form the essence of the SER model:

- On the *conceptual framework level*, the multifaceted, domain-independent architecture constitutes a framework for building evolvable complex software systems.
- When this architecture is instantiated in a domain (e.g., network design), a domain-oriented design environment (representing an application family) is created on the *domain level*. An instantiation in the network domain is NetDE.
- Individual artifacts in the domain are developed by exploiting the information contained in the generic DODE (in the scenario this is represented by the network developed for the Publications Group).

Fig. 3 illustrates the interplay of those three layers. Darker gray indicates knowledge domains close to the computer, whereas white emphasizes closeness to the design work in a domain. The figure illustrates the role of different professional groups in the evolutionary design: the *environment developer* (close to the computer) provides the domain-independent framework, and instantiates it into a DODE in collaboration with the help of the domain designers (knowledgeable domain workers who use the environment to design artifacts; in the scenario, D_1 is a domain designer). Domain designers are the "end users" of a design environment. The artifact is eventually delivered to the client (e.g., the Publications Group in the scenario).

Breakdowns occur when domain designers cannot carry out the design work with the existing DODE. Extensions and criticism drive the evolution on all three levels: Domain designers directly modify the artifacts when they build them (artifact evolution), they feed their modifications back into the environment (domain evolution), and—during a reseeding phase—even the architecture may be revised (conceptual framework evolution). In Fig. 3, the little building blocks represent knowledge and domain elements in any of the components of the multifaceted architecture (i.e., the Indy, critics about network protocols, etc.).

The evolution of complex systems in the context of this process model (more detail can be found in [20]) can be characterized as follows:

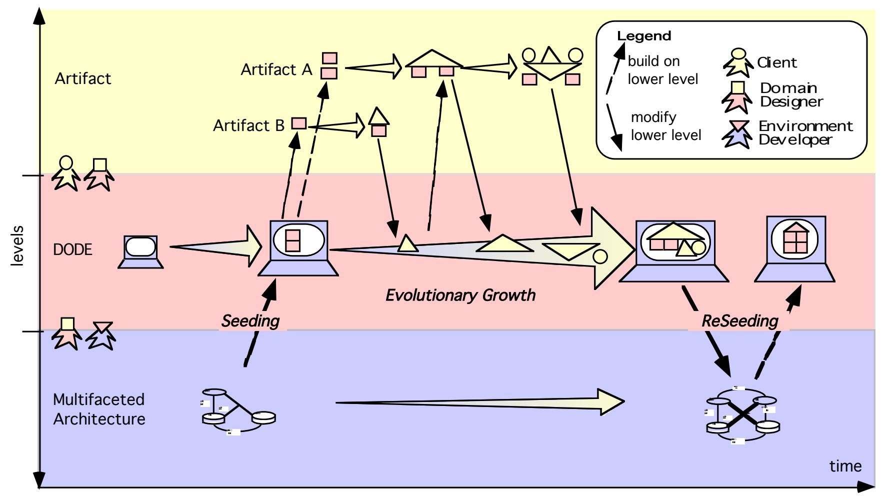


Fig. 3. The SER Model: A process model for the development and evolution of DODEs

Seeding. A seed will be created through a participatory design process between environment developers and domain designers. It will evolve in response to its use in new network design projects because requirements fluctuate, change is ubiquitous, and design knowledge is tacit. Postulating the objective of a seed (rather then a complete domain model or a complete knowledge base) sets our approach apart from other approaches in software engineering and artificial intelligence and emphasizes evolution as the central design concept.

Evolutionary growth. Network experts use the seeded environment to undertake specific projects for clients (such as the Publication Group in the scenario). During these design efforts, new requirements may surface (e.g., the desire to access the Internet), new components may come into existence (e.g., the Indy) and additional design knowledge not contained in the seed may be articulated (e.g., that EtherTalk supports both AppleTalk and TCP/IP protocols). During the evolutionary growth phase, the environment developers are not present, thus making end-user modification a necessity rather than a luxury (at least, as argued before, for small-scale evolutionary changes). Visual AgenTalk (VAT) addresses this problem, building on our experience with end-user modifiability and end-user programming [13,17].

Reseeding. In a deliberate effort to revise and coordinate information and functionality, the environment developers are brought back in to collaborate with domain designers to organize, formalize, and generalize knowledge added during the evolutionary growth phases. Organizational concerns [24] 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.

Drastic and large-scale evolutionary changes occur during the reseeding phase.

5 Systems-Building Efforts

The preceding sections have identified important issues that need to be addressed to support evolutionary system development: design rationale, design-in-use, end-user modifiability, and collaboration support for a community of practice. Following prototypical systems are briefly introduced that address those issues.

5.1 Group Interactive Memory Manager (GIMMe)

The research objective behind GIMMe [19] is to support the communication between different stakeholders involved in the development process and to capture and structure this communication in order to make it immediately available as design rationale [32].

Description. GIMMe is a web-based group memory system. It helps communities of practice (e.g., project teams, interest groups) to capture, store, organize, share, and retrieve electronic mail conversations. Mail sent to a specific group alias is automatically added to an information space and categorized according to its subject line. Group members can access the information space via the Internet. It supports three retrieval mechanisms to this information space: (1) browsing in reverse chronological order, (2) browsing according to project-specific categories, and (3) retrieval by free-form text queries (using the Latent Semantic Indexing algorithm [27]. GIMMe supports users in creating, rearranging, or deleting categories and the mail belonging to them (a more detailed description of GIMMe can be found at http://www.cs.colorado.edu/~stefanie). It will allow groups to create and negotiate domain conventions and concepts over time, as well as to evolve categories that reflect the structure and vocabulary of the application domain.

How does GIMMe support evolution? In order for the users to evolve a system they have to be able to understand the design decisions that lead to the current system. GIMMe addresses the problem of how to capture this important information, structure it for later reuse, and make it available to the users. GIMMe is an effort demonstrating that design rationale emerges as a by-product of normal work. This is critical because we know from empirical evidence that most design rationale systems have not failed because of the inadequacy of a computational substrate, but because they did not pay enough attention to the question "who is the beneficiary and who has to do the work?" [24]. Our experiences with the use of GIMMe at NYNEX Science & Technology and the University of Colorado have encouraged us to pursue this idea of a growing group memory and have shown that such a system can be employed for real world problems and projects. An evolvable DODE and evolvable artifacts within a DODE will require design rationale and, therefore, GIMMe has to be an integral component.

5.2 Expectation Agents

Expectation Agents [22] were designed to support communication between end users and developers of an interactive system during actual use situations [25]. Expectation Agents observe and analyze the reactions of end users to the system by not relying on only a small subset of end users but by reaching the whole (or a large subset) of the community of practice.

Description. Expectation Agents are an active part of the system in which the end user designs artifacts. They observe the actions of the end user and compare them to descriptions of the intended use of the system. These descriptions are provided by the system developers and represent their expectations about how the system should be used (e.g., in which order certain tasks are performed). If an Expectation Agent identifies a discrepancy between how an end user uses the system and the description provided by a system developer it then notifies the developer and prompts the end user for an explanation. This explanation is then emailed to the developer as well, establishing a communication between the system developer and a specific end user.

How do Expectation Agents support evolution? Expectation Agents are used to support evolutionary growth. When developers and users create a seed (see Fig. 3) they hold a number of explicit and implicit assumptions about how the system will be used and how it supports the work practices. Expectation Agents are one way to compare those assumptions to actual use patterns.

5.3 Visual AgenTalk (VAT)

The objective behind VAT is to support end-user modifications and programming as an essential component for an evolvable DODE [13].

Description. VAT [40] (a detailed description of VAT can be found at http://www.cs.colorado.edu/~ralex) has been created on top of the Agentsheets programming substrate. Agentsheets is used as a substrate for the construction component (e.g., specifically to create interactive graphical simulations of complex dynamic systems). These simulations consist of active agents that interact with each other and exhibit a specific behavior. Combined with Agentsheets in a layered environment, Visual AgenTalk is used to define end-user programming languages that are tailorable to a particular domain, promote program comprehensibility, and provide end users with control over powerful, multimodal interaction capabilities. VAT provides mechanisms for the creation of commands so that domain-specific languages can be developed. Conditions, actions, and rules are all graphical objects, and end users can try out their programs by dragging and dropping them onto agents in a worksheet. The ability to test programs within the context of a particular agent increases the end user's ability to comprehend Visual AgenTalk programs.

How does VAT support evolution? VAT enables end users to modify and program the behavior of active agents inside a simulation environment. New agent types can be created, modified, and shared. Thus VAT promotes evolutionary growth on the hands of end users. VAT extends the construction and simulation component in order to accommodate end-user programming. VAT is used to implement expectation agents, thereby giving end users the possibility to modify the behavior of expectation agents. Agentsheets provides mechanisms that can be used by both environment developers and domain designers to define the look of individual construction objects, and VAT can be used by environment developers to define domain-specific languages to be used by domain designers to evolve the behavior of the construction objects.

5.4 New Conceptualization of the World Wide Web

The Web in its current form does not support evolutionary design. Fig. 4 presents three models that illustrates different types of Web usage [3].

Traditional Web-based use engages the Web as a Broadcast Medium (Fig. 4, Model M_1). In this model, instructional content is predetermined and placed on static Web pages. Most popular general-purpose Web tools provide support for the easy generation of this static content. In M_1, the Web serves as a distribution channel and provides few opportunities for learners to interact with the information because the content was not originally designed to be interactive. Responding to the need for feedback from consumers, many Web sites are evolving into forms that augment content with some communication

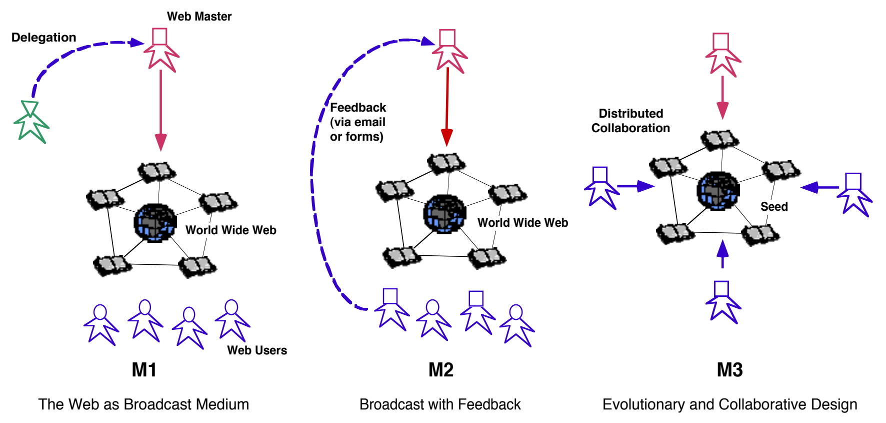

Delegation
Web Master
World Wide Web
Web Users
M1
The Web as Broadcast Medium
Feedback (via email or forms)
World Wide Web
M2
Broadcast with Feedback
Distributed Collaboration
Seed
M3
Evolutionary and Collaborative Design

Fig. 4. Making the World Wide Web a Medium for Collaborative, Evolutionary Design

channels. This mechanism of *broadcast with feedback* (M_2) expands the original model by providing some link from consumer to producer such as allowing learners to provide feedback and ask questions by filling out forms. Although users can react to information provided by the author, this presentation model provides little support for evolution.

The M_3 model demonstrates an essential requirement for collaboration and evolution for the Web. In M_3, users can use the Web to collaborate on projects by *actively* contributing and by learning from all contributors. The evolution of content and ideas is now the responsibility of the participating community of practice, focusing on the distributed generation of content and the reflection upon it. An M_3-type model is needed to support the SER model. When a wide variety of individuals collaborate in a cooperative forum, the unique skills of the members all become valuable resources in making the Web resources useful in the current context. The M_3 model poses a number of technical challenges, including the ability (1) to add to an information space without going through an intermediary, (2) to modify the structure of the information space, and (3) to modify at least some of the existing information.

6 Assessment

6.1 Understanding Pitfalls Associated with Evolutionary Design

To make evolutionary design a more ubiquitous activity, the forces that prohibit or hinder evolution must be understood. Examples of such forces are: (i) the resistance to change because it requires learning efforts and may create unknown difficulties and pressures, (ii) the problem of premature standards, (iii) the difficulties created by installed bases and legacy systems, and (iv) the issues of who are the beneficiaries and who has to do the work in order for evolution to occur.

The Oregon Experiment [2] (a housing experiment at the University of Oregon instantiating the concept of end user-driven evolution) serves as an interesting case study that end user-driven evolution is no guarantee for success. The analysis of its unsustainability indicated the following major reasons: (1) there was a lack of continuity over time, and (2) professional developers and users did not collaborate, so that there was a lack of synergy. These findings led us in part to postulate the need for a reseeding phase (making evolutionary development more *predictable*), in which developers and users engage in intense collaborations. With design rationale captured, communication enhanced, and end user support available, developers have a rich source of information to evolve the system in the way users really need it. Another interesting source of information for the SER model is Kuhn [26], in which general conceptual frameworks can be found to decide when the time has come to engage in a reseeding process rather than continue with evolutionary growth.

6.2 Assessment of the SER Model

The SER model is motivated by how large software systems, such as Emacs, Microsoft-Word, Unix, and the X Window System, have evolved over time. 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 (following the observation that any artifact should be useful in the expected way, but a truly great artifact lends itself to uses the original designers never expected). New releases of the system often incorporate ideas and code produced by users. In the same way that these software systems are extensible by programmers who use them,

Open-Source Software Systems. The development of the Linux operation system [37] provides an interesting existence proof that reliable, useful, and usable complex systems can be built in a decentralized "Bazaar style" by many [41] rather than in a centralized, "Cathedral style" by a few. The Linux development model treats users as co-developers and is currently tested in a number of new areas, such as: (1) *Netscape Communicator* (for more information see http://www.mozilla.org/); (2) Gamelan (http://www.gamelan.com; the first community repositories of Java-related information allowing Java developers looking for information about what other people are doing with Java; the large number of developers who contribute to the Gamelan repository and the number of people who search for information in Gamelan provide evidence that the Java community has taken a great deal of interest in using community repositories to locate information); (3) *Educational Object Economy* (EOE; http://trp.research.apple.com/; the EOE is realized as a collection of Java objects (mostly completed applets) designed specifically for education; the target users of the EOE are teachers wishing to use new interactive technology and developers interested in producing educational software).

Domain-Oriented Design Environments. DODEs poses a major additional challenge to make the SER model feasible and workable: Whereas the people in the above mentioned development environments are computationally sophisticated and experienced users, DODEs need to be extended by domain designers who are neither interested in nor trained in the (low-level) details of computational environments. The SER model explores interesting new ground between the two extremes of "put all the knowledge in at the beginning" and "just provide an empty framework." Designers are more interested in their design task at hand than in maintaining a knowledge base. At the same time, important knowledge is produced during daily design activities that should be captured. Rather than expect designers to spend extra time and effort to maintain the knowledge base as they design, we provide tools to help designers record information quickly and without regard for how the information should be integrated with the seed. Knowledge-base maintenance is periodically performed during the reseeding phases by environment developers and domain designers in a collaborative activity.

7 Evolutionary Design—Beyond the Boundaries of Disciplines

7.1 Avoid Reinventing the Wheel

In this article, I have mostly discussed examples from the domain of software design. Software design is a new design discipline relative to other more established disciplines. Software designers can learn a lot by studying other design disciplines such as architectural design, graphic design, information design, urban 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 [12]. 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.

7.2 Evolutionary Design in Architecture

Evolutionary design is a concept of equal important in architecture [6]. For many arguments and considerations articulated in this article, the words "software systems" and "buildings", and "software designer/programmer" and architect are interchangeable. Software design being the much younger discipline could learn a lot from design methodologies developed in architectural design. Designing complex artifacts from scratch, while often considered to be highly desirable to avoid the constraints of dealing with existing structures and legacy systems, leads to artifacts which are often missing a "quality" that exists in evolving artifacts—as illustrated by cities such as Brasilia and Abuja versus cities such as London and Paris. Artifacts are embedded in time, and over time many of the determining factors influencing a design will change; and because buildings and cities are going to be modified many times, they should be designed with unanticipated future changes in mind.

The problems facing both professional groups have initiated at least some interest in each other work. Alexander's work [1] on patterns has found many followers in the software design community, specifically in object-oriented design [21]. In our own research we have created an "Envisionment and Discovery Collaboratory" to explore new computational environments that enhance communication between different stakeholders, facilitate shared understanding, and assist in the creation of better artifacts by integrating physical and computational media for design [4]. By doing so, we attempt to integrate the best of both worlds: the dynamic nature of computational media and the strength of physical media in allowing people to operate and think with tangible objects.

7.3 DODEs in Architecture

The concept of DODEs can be applied to many areas, and in our work (in close collaborations with professionals from the respective design disciplines), we have created DODES for kitchen design [14], lunar habitat design [46], and urban design [4]. Design activities embedded in computational environments are the best domains for DODEs, because the activities take place *within* the computational environment and the power of DODEs (providing critics, linking action and reflection spaces, supporting simulations, etc.) can be most successfully and most easily exploited.

8 Conclusions

DODEs are software systems that support design activities within a particular domain and are built specifically to evolve. DODEs have provided the foundations in our research to develop a theoretical and conceptual framework for the evolutionary design of complex systems illustrating (i) the importance of end user modifiability and end user programming, (ii) the capture and retrieval of design rationale, and (iii) the necessity of improved communications among different design stakeholders including system developers and end users.

Evolution of complex systems is a ubiquitous phenomenon. This is true in the physical domain, where, for example, artificial cities such as Brasilia are missing essential ingredients from natural cities such as London or Paris. "Natural" cities gain essential ingredients through their evolution—designers of "artificial" cities are unable to anticipate and create these ingredients. It is equally true for software systems for the reasons argued in this paper. A challenge for the future is to make (software) designers aware of essential concepts that originated and were explored in evolution, such as ontogeny, phylogeny, and punctuated equilibrium. Even though we are convinced that models from biology may be more relevant to future software systems than models from mathematics, we also have to be cautious: to follow an evolutionary approach in software design *successfully* does not imply that concepts from biological evolution should be mimicked literally, but rather they need to be reinterpreted in the domain of software design [5].

References

1 Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; Angel, S., *A Pattern Language: Towns, Buildings, Construction*; Oxford University Press: New York, 1977.

2 Alexander, C.; Silverstein, M.; Angel, S.; Ishikawa, S.; Abrams, D., *The Oregon Experiment*; Oxford University Press: New York, NY, 1975.

3 Ambach, J.; Fischer, G.; Ostwald, J.; Repenning, A., *Making the World Wide Web A Medium for Collaborative, Evolutionary Design*, At http://www.cs.colorado.edu-/~ostwald/papers/WWW97/PAPER200.html, 1997.

4 Arias, E. G.; Fischer, G.; Eden, H., *Enhancing Communication, Facilitating Shared Understanding, and Creating Better Artifacts by Integrating Physical and Computational Media for Design*, In *Proceedings of Designing Interactive Systems (DIS '97)*; Amsterdam, The Netherlands, 1997; pp. 1-12.

5 Basalla, G., *The Evolution of Technology*; Cambridge University Press: New York, 1988.

6 Brand, S., *How Buildings Learn—What Happens After They're Built*; Penguin Books: New York, 1995.

7 Brooks, F. P., Jr., *No Silver Bullet: Essence and Accidents of Software Engineering*, In *IEEE Computer* 1987, *20*, pp. 10-19.

8 Computer Science and Technology Board, *Scaling Up: A Research Agenda for Software Engineering*, In *Communications of the ACM* 1990, *33*, pp. 281-293.

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

10 Curtis, B.; Krasner, H.; Iscoe, N., *A Field Study of the Software Design Process for Large Systems*, In *Communications of the ACM* 1988, *31*, pp. 1268-1287.

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

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

13 Eisenberg, M.; Fischer, G., *Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance*, In *Human Factors in Computing Systems, CHI'94*; Boston, MA, 1994; pp. 431-437.

14 Fischer, G., *Domain-Oriented Design Environments*, In *Automated Software Engineering* 1994, *1*, pp. 177-203.

15 Fischer, G., *Putting the Owners of Problems in Charge with Domain-Oriented Design Environments* In *User-Centred Requirements for Software Engineering Environments*; R. W. D. Gilmore, F. Detienne, Ed.; Springer Verlag: Heidelberg, 1994; pp. 297-306.

16 Fischer, G., *Turning Breakdowns into Opportunities for Creativity*, In *Knowledge-Based Systems, Special Issue on Creativity and Cognition* 1995,

17 Fischer, G.; Girgensohn, A., *End-User Modifiability in Design Environments*, In *Human Factors in Computing Systems, (CHI'90)*; Seattle, WA, 1990; pp. 183-191.

18 Fischer, G.; Lemke, A. C.; McCall, R.; Morch, A., *Making Argumentation Serve Design* In *Design Rationale: Concepts, Techniques, and Use*; T. Moran and J. Carrol, Ed.; Lawrence Erlbaum and Associates: Mahwah, NJ, 1996; pp. 267-293.

19 Fischer, G.; Lindstaedt, S.; Ostwald, J.; Schneider, K.; Smith, J., *Informing System Design Through Organizational Learning*, In *International Conference on Learning Sciences (ICLS'96)*; Chicago, IL, 1996; pp. 52-59.

20 Fischer, G.; McCall, R.; Ostwald, J.; Reeves, B.; Shipman, F., *Seeding, Evolutionary Growth and Reseeding: Supporting Incremental Development of Design Environments*, In *Human Factors in Computing Systems (CHI'94)*; Boston, MA, 1994; pp. 292-298.

21 Gamma, E.; Johnson, R.; Helm, R.; Vlissides, J., *Design Patterns - Elements of Reusable Object-Oriented Systems*; Addison-Wesley: Reading, MA, 1994.

22 Girgensohn, A.; Redmiles, D.; Shipman, F., *Agent-Based Support for Communication between Developers and Users in Software Design* In *Proceedings of the 9th Annual Knowledge-Based Software Engineering (KBSE-94) Conference (Monterey, CA)*; IEEE Computer Society Press: Los Alamitos, CA, 1994; pp. 22-29.

23 Greenbaum, J.; Kyng, M., *Design at Work: Cooperative Design of Computer Systems*; Lawrence Erlbaum Associates, Inc.: Hillsdale, NJ, 1991.

24 Grudin, J., "Seven plus one Challenges for Groupware Developers," 1991.

25 Henderson, A.; Kyng, M., *There's No Place Like Home: Continuing Design in Use* In *Design at Work: Cooperative Design of Computer Systems*; J. Greenbaum and M. Kyng, Ed.; Lawrence Erlbaum Associates, Inc.: Hillsdale, NJ, 1991; pp. 219-240.

26 Kuhn, T. S., *The Structure of Scientific Revolutions*; The University of Chicago Press: Chicago, 1970.

27 Landauer, T. K.; Dumais, S. T., *A Solution to Plato's Problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge*, In *Psychological Review* 1997, *104*, pp. 211-240.

28 Laszlo, E., *Evolution: The Grand Synthesis*; Shambhala Publications, Inc.: 1987.

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

30 Lewis, T., *Object-Oriented Application Frameworks*; Prentice Hall: Englewood Cliffs, New Jersey, 1995, 344 pages.

31 MacLean, A.; Carter, K.; Lovstrand, L.; Moran, T., *User-Tailorable Systems: Pressing the Issues with Buttons* In *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*New York, 1990; pp. 175-182.

32 Moran, T. P.; Carroll, J. M., *Design Rationale: Concepts, Techniques, and Use*; Lawrence Erlbaum Associates, Inc.: Hillsdale, NJ, 1996.

33 Nardi, B.; Zarmer, C., *Beyond Models and Metaphors: Visual Formalisms in User Interface Design*, In *Journal of Visual Languages and Computing* 1993, pp. 5-33.

34 Norman, D. A., *Turn Signals are the Facial Expressions of Automobiles*; Addison-Wesley Publishing Company: Reading, MA, 1993.

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

36 Popper, K. R., *Conjectures and Refutations*; Harper & Row: New York, Hagerstown, San Francisco, London, 1965.

37 Raymond, E. S., *The Cathedral and the Bazaar*, At http://earthspace.net/~esr/writings-/cathedral-bazaar/cathedral-bazaar.html, 1998.

38 Redmiles, D. F., “From Programming Tasks to Solutions—Bridging the Gap Through the Explanation of Examples,” 1992.

39 Repenning, A.; Ambach, J., *The Agentsheets Behavior Exchange: Supporting Social Behavior Processing*, In *Computer-Human Interaction (CHI '97)*; Atlanta, GA, 1997; pp. 26-27 (Extended Abstracts).

40 Repenning, A.; Ioannidou, A., *Behavior Processors: Layers between End-Users and Java Virtual Machines*, In *Visual Languages*; Capri, Italy, 1997; pp. 402-409.

41 Resnick, M., *Turtles, Termites, and Traffic Jams*; The MIT Press: Cambridge, MA, 1994.

42 Rich, C. H.; Waters, R. C., *Automatic Programming: Myths and Prospects* In *Computer*; The Computer Society: Los Alamitos, CA, 1988; Vol. 21; pp. 40-51.

43 Rittel, H., *Second-Generation Design Methods* In *Developments in Design Methodology*; N. Cross, Ed.; John Wiley & Sons: New York, 1984; pp. 317-327.

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

45 Simon, H. A., *The Sciences of the Artificial*; The MIT Press: Cambridge, MA, 1996.

46 Stahl, G., *Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design*, Ph.D. dissertation. UMI#9423544, Department of Computer Science. University of Colorado at Boulder. Technical Report CU-CS-688-93, 1993.

47 Swartout, W. R.; Balzer, R., *On the Inevitable Intertwining of Specification and Implementation* In *Communications of the ACM*, 1982; Vol. 25; pp. 438-439.

48 Terveen, L. G.; Selfridge, P. G.; Long, D. M., *Living Design Memory: Framework, Implementation, Lessons Learned*, In *Human-Computer Interaction* 1995, *10*, pp. 1-37.

49 Winograd, T., *From Programming Environments to Environments for Designing*, In *Communications of the ACM* 1995, *38*, pp. 65-74.