University of Colorado at Boulder

Center for LifeLong Learning and Design (L3D)

Department of Computer Science

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

From Domain Modeling to Collaborative Domain Construction

Gerhard Fischer[1], Stefanie Lindstaedt[1,2], Jonathan Ostwald[1,2], Markus Stolze[1], Tamara Sumner[1] and Beatrix Zimmermann[2]

[1]Department of Computer Science and the
Center for LifeLong Learning and Design
University of Colorado at Boulder
Boulder, CO 80309-0430
gerhard@cs.colorado.edu

[2]NYNEX Science & Technology (S&T)
500 Westchester Avenue
White Plains, NY 10604
bz@nynexst.com

DIS'95 (Ann Arbor, MI), 1995 ACM, pp. 75- 85

From Domain Modeling to Collaborative Domain Construction

Gerhard Fischer[1], Stefanie Lindstaedt[1,2], Jonathan Ostwald[1,2], Markus Stolze[1], Tamara Sumner[1], Beatrix Zimmermann[2]

[1] Department of Computer Science and the Center for LifeLong Learning and Design
University of Colorado
Boulder, CO USA 80309-0430
Email: (gerhard, stefanie, ostwald, stolze, sumner)@cs.colorado.edu

[2] NYNEX Science and Technology
500 Westchester Avenue
White Plains, NY 10604
Email: bz@nynexst.com

ABSTRACT

Domain-oriented systems offer many potential benefits for end-users such as more intuitive interfaces, better task support, and knowledge-based assistance. A key challenge for system developers constructing domain-oriented systems is determining what the current domain is and what the future domain should be; i.e. what entities should the system embody and how should they be represented. Determining an appropriate domain model is challenging because domains are not static entities that objectively exist, but instead they are dynamic entities that are constructed over time by a community of practice. New software development models and new computational tools are needed that support these communities to create initial models of the domain and to evolve these models over time to meet changing needs and practices. We describe a specific software development model and computational tools that enable domain practitioners to participate in domain construction processes.

KEYWORDS: software design, domain-oriented design environments, design, domain modeling, domain construction

INTRODUCTION

Orienting software systems towards specific domains or tasks has been heralded by many researchers as a means of making software both more useful and more usable [9, 11, 20, 21, 29]. Domain-oriented software is more usable than generic software because users directly interact with familiar entities and do not need to learn new computer-specific concepts [11]. It is more useful than generic software because the provided functionality directly targets tasks relevant to the domain and users do not have to build up desired behaviors from other lower-level operations [17]. In our work, we have created numerous domain-oriented design environments and domain-oriented visual programming languages for domains such as kitchen design, network design, voice dialog design, and telephone service provisioning [8, 18, 25, 27, 28].

DIS 95 Ann Arbor MI USA © 1995 ACM 0-89791-673-5/95/08..$3.50

Domain-oriented systems embody a model of the entities to be manipulated and the tasks to be performed. Spreadsheets, one of the most commercially successful software applications of all time, are often cited as an example of a system oriented towards the domain of financial modeling and accounting. The domain model provided by spreadsheets includes numerical entities such as monetary values and dates, a tabular representation for viewing these entities, and operations for manipulating the entities such as statistical and financial functions. In our work, we have focused on providing richer domain models than those found in commercial systems such as spreadsheets. These rich models allow our domain-oriented design environments to provide knowledge-based assistance to users engaged in ill-structured design tasks [10, 19].

While, on the whole, the outlook for domain-oriented systems is optimistic, they are not without their drawbacks. First, determining what the domain model should be and constructing the model is a difficult and time-consuming process. Second, the world is not static; what should be in the domain model changes over time as technology and practices evolve. Finally, many domain-oriented systems are too rigid and limited to support the rich set of tasks that professional practitioners need to perform. Nardi found that professional slide makers preferred generic graphic tools to slide making-specific software for this very reason [22]. *In our view, these drawbacks to domain-oriented systems can be avoided or minimized by moving the system development process from a domain modeling approach towards a collaborative domain construction approach.*

The domain modeling approach assumes there exists a common conceptual model of the domain shared by all practitioners and the problem is simply to identify what this model is and codify it [24]. This approach falls into the category of first generation design methods [6, 30] that assume a strict separation between design, use, and implementation phases. In the design phase, these approaches try to identify the domain model through knowledge acquisition practices [3] such as interviewing selected domain experts. As such, these approaches do not acknowledge the situated [32] and tacit [23] nature of

professional expertise. In the implementation phase, these approaches adopt an engineering perspective in that they emphasize domain model representations rooted in computational formalisms rather than representations rooted in work practices. The result is domain models that cannot be inspected or modified by domain practitioners to reflect changing work practices.

The domain construction approach address these shortcomings by explicitly acknowledging that shared domain models do not de facto exist but instead are socially constructed over time by communities of practice. As such, this approach emphasizes the prominent role of communication and mutual learning between domain practitioners and system developers both in constructing an initial model of the domain rooted in domain practice, and in evolving this model over time to suit the changing needs of practitioners. Mutual learning is required because developers need to learn about the current domain and practitioners need to learn how current practices might be transcended with new technologies. Computational tools and new models of software development are needed that promote communication and mutual learning by all design stakeholders throughout the design and evolution of their domain-oriented system.

In this paper, we will present software development models and tools that support mutual learning during the domain construction process. We begin this paper by illustrating how domains are socially constructed over time with a story drawn from our empirical investigations of design practices. Next, we describe a new model for software development and discuss how this model supports the social construction and evolutionary aspects of domains. Finally, we present software tools that we have used to create various domain-oriented systems and assess how well these tools support the domain construction process.

EXAMPLE: CONSTRUCTING THE VOICE DIALOG DESIGN DOMAIN

A domain consists of the set of objects, representations, and practices capable of expressing important distinctions in the range of problems that practitioners need to solve. As such, domains are actively created by domain practitioners from insights gained and breakdowns in reoccurring work activities [35]. In this section, the experiences of a particular set of domain practitioners are presented – voice dialog designers at a regional phone company. This story is an excerpt from a long-term case study of these designers' work practices [33]. The point of the story is twofold. First, it illustrates how a diverse group of design stakeholders collectively constructed their domain over a three year period. Second, it illustrates how domain practitioners and system developers can develop tools to support new ways of working by viewing the system design process as one of collaborative domain construction.

Social Domain Construction

Voice dialog designers create software applications that have phone-based user interfaces. Typical applications are voice information systems and voice messaging systems. These applications consist of a series of voice-prompted menus requesting the user to perform certain actions; e.g., "to listen to your messages, press 1." The caller issues commands by pressing touch-tone buttons on the telephone keypad and the system responds with appropriate voice phrases and prompts.

Designing in this domain means specifying the interface for an application at a detailed level. There are two main facets to the designer's job: constructing the design and communicating the evolving design to other design stakeholders such as marketing and the vendor organization. Towards this end, the designers have created different design representations tailored to the special needs of each of the major stakeholder groups. Flow charts are the primary design representation and are constructed to communicate the essential aspects of the interface such as spoken prompts, menus, and control flow to the marketing and vendor organizations. Additional detailed design information is presented in a separate table representation that is primarily constructed for the vendor organization.

However, as Figure 1 shows, both the representations and the objects these representations depict have considerably evolved over time. As these practitioners worked, they constructed their domain by refining existing objects and representations and by creating new objects and representations. Sometimes these changes were introduced simply to improve operations; other times they were introduced to overcome specific problems in their design activities.

The design languages; i.e., the objects and representations, used by this community went through a maturation process as they progressively evolved from being ill-defined to well-defined. The original textual specifications (Figure 1, left side) were ill-defined because important aspects of the domain were not made explicit by the representation. Important interface features were buried in the middle of paragraphs and this resulted in design errors and communication breakdowns. As designers recognized common breakdowns in the design process, they created objects and representations to overcome these breakdowns. For instance, the initial table representations (Figure 1, middle) were created to satisfy the vendor organization that needed to easily see all possible actions the system was to perform in response to user input. One lead designer became concerned that the marketing organization was no longer bothering to read the textual specifications due to their complexity. He pioneered the use of flow charts and tables (Figure 1, right side) as the primary design representations. These representations have changed significantly since their initial introduction as designers have tried to improve their visual clarity and as other stakeholders have suggested further refinements. These representations are well-defined because they make more explicit significant domain objects and their relationships. Because these well-defined representations were socially constructed by all design stakeholders, their interpretation is shared and standardized.

Supremacy of textual specifications

When the mailbox is entered the system will play, "PLEASE ENTER YOUR SECURITY CODE NOW."

If the user does not respond, the system plays, *"are you still there?* PLEASE CHECK YOUR SECURITY CODE AND ENTER IT NOW."

If the user enters an invalid code, the system plays, *"I'm sorry that is not the correct security code."*

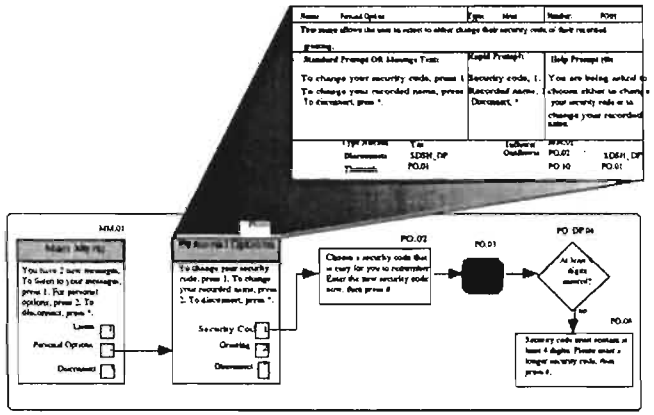
If the user presses *, the system backs out to open trees.

Augmentation of text with simple tables and flow charts

Phrase (**B**): "Please enter your security code now."

ACTION	EXPECTED RESPONSE	GO TO
timeout	Are you still there? Please...	**B.1**
invalid	I'm sorry that is not the ...	**B.1**
press *		**A**

Emergence of elaborate flows and tables as preferred representational system



Pre 1991 **Now**

77

As time progresses, more objects populate the domain and are incorporated into increasingly formal representations.

Two types of system responses are distinguished: phrases (all caps) and messages (italics). Possible user actions are described in indented text. Control flow is described using natural language.

These tables depict possible actions after phrases request users to do something. Phrases are assigned identifiers which are bold letter/number combinations. The identifiers are interspersed into the text to tie it to the tables.

The phrase tables have evolved into two distinct entities: prompts and voice menus. Messages are still represented distinctly. New objects such as decision points and data inputters are now represented in the flow charts. Control flow is depicted graphically with links. Every object in the flow chart has an identifier that ties it to a corresponding table representation.

Figure 1: Construction of the Voice Dialog Design Domain

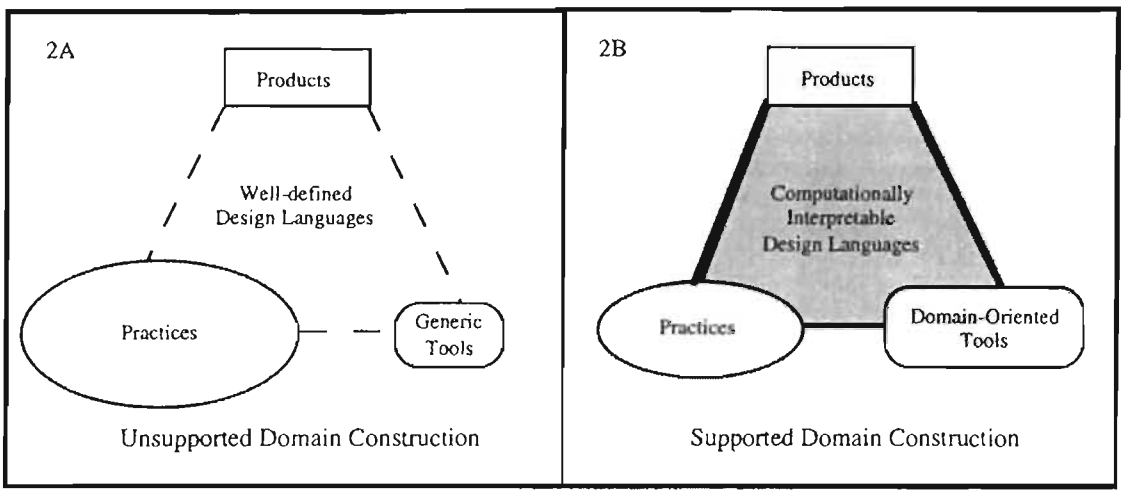


Figure 2: Unsupported versus Supported Domain Construction

With unsupported domain construction (A), it is practices that are enriched and serve to perpetuate well-defined design languages. With supported domain construction (B), tools can also be enriched with domain knowledge. The resulting computationally interpretable design language serves to tightly couple design practices, tools, and products.

However, it would be a mistake to assume that these current flow chart and table representations contain all relevant domain objects and relationships; i.e. that these are the final "picture" and will never change. There are many aspects of this domain that are still being developed and will be furthered refined as the result of ongoing design activities. In this sense, the maturation process is never complete but instead is ongoing and continuous.

The voice dialog design story illustrates how design languages are socially constructed over time as important objects and their relationships emerge and are incorporated into design representations. We call this process "domain construction." While the maturing of the voice dialog design languages definitely benefited the designers' overall design activities, this domain construction process was not without some cost. The problem is that the domain construction process is not supported by the designers' software tools.

Currently, these designers use generic software tools such as word processors, databases, and flow charting packages to construct the necessary design representations. These tools are "generic" because they have no built in knowledge of the voice dialog domain. Sumner [33] documented how the tools' lack of domain knowledge adversely impacted design practices. As design languages mature, there are more domain objects represented with greater levels of detail and more dependencies and relationships between these objects across design representations. These details and dependencies must be maintained so that the various representations are consistent. Since the tools are generic; i.e., they have no knowledge that voice menu PO.01 in the flow chart is related to voice menu PO.01 in the table representation, the designers must manually manage these details and dependencies themselves. Besides being tedious and error prone, the manual and cognitive burdens of managing these dependencies hindered the designers' beneficial iterative design process by making iteration slower and more costly.

Figure 2a illustrates how in the voice dialog design situation described above, the designers' tools, products, and practices are loosely coupled by well-defined design languages; we call this "unsupported domain construction." The coupling is loose and unsupported because it is largely the designers' practices that have evolved to incorporate their understanding of the domain and the tools remain unenriched with any domain knowledge.

Transcending Current Practices

We claim that a more desirable outcome is for tools to be enriched with some understanding of the domain. This enriching would: (1) allow tools to more directly support designers' practices by alleviating some of the manual and cognitive burdens detailed in [33], and (2) enable new work practices not possible with generic tools.

We call this enriching process "supported domain construction," where the designers' tools, products, and practices are tightly coupled by computationally interpretable design languages (Figure 2b). Computationally interpretable languages are shared by domain practitioners and their tools. Enriching tools with interpretable languages is the next step in an ongoing domain maturation process. These languages differ from traditional domain models created with first generation domain modeling techniques in that their representation is firmly rooted in the work practices of the practitioners rather than in the world of computational formalisms. As such, interpretable design languages are meant to be inspected and changed by domain practitioners in order to support the ongoing process of domain construction.

The Voice Dialog Design Environment (VDDE) is an example of a system with a computationally interpretable

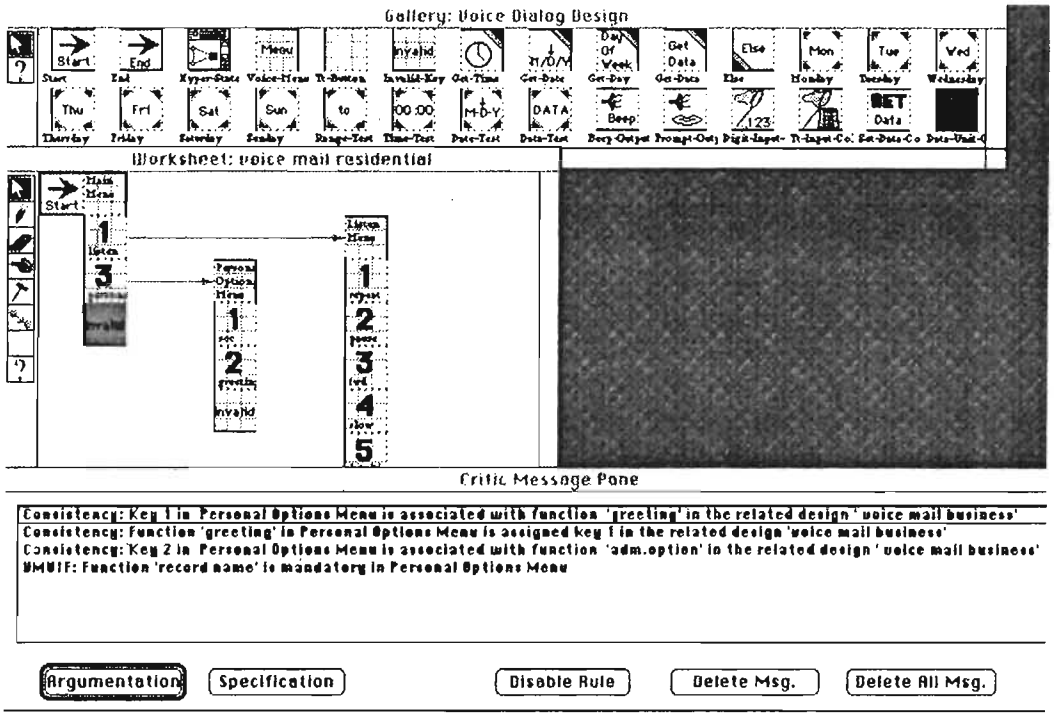


Figure 3: The Voice Dialog Design Environment

The computationally interpretable design language supported by the VDDE system is based on the flow chart design representation. The VDDE system transcends current practices by allowing designers to simulate their design construction at any time and to critique their design for usability and consistency with the existing product line.

design language (Figure 3). VDDE is the result of a collaborative design process between system developers and the user interface designers described above [29, 34]. One goal of the VDDE project was to provide a design environment that enabled user interface designers to quickly create their own simulations of the audio interfaces being designed. Design simulations play a crucial role in the voice dialog design process because they allow domain practitioners and their customers to directly experience the final audio interface. Unfortunately, using current software packages, a simulation for a simple design takes a professional programmer several days to build; a simulation for a complex design can take a couple of weeks to produce.

The design language provided by VDDE includes: (1) a gallery containing objects such as prompts, messages and voice menus, (2) a worksheet for constructing flow chart-like design representations, (3) a simulation component that executes the flow chart to produce an audio presentation of all prompts and messages, and (4) knowledge bases of design rules that critique the designs for usability and consistency with the existing product line. This language illustrates a first step towards supported domain construction. Informal tests of the new design language indicate that simulations that once took designers several days to build can now be created in a couple of hours.

The core of the VDDE language is based on the flow chart design representation and is thus rooted in the designers' work practices. However, this language transcends the flow chart representation in that it can be executed to produce voice output. Supporting design simulation required the VDDE language to introduce new domain objects not present in the flow chart. This language was refined through a series of collaborative design sessions between domain practitioners and system developers. During these sessions, successive versions of the language were used by practitioners to solve existing design problems and language extensions were driven by breakdowns in use situations [29].

However, the current VDDE system only partially satisfies the requirements for supported domain construction. First, while VDDE is an extensible system that has evolved, the programming substrate VDDE is built on requires programming knowledge to add new objects or modify existing ones. Thus, VDDE does not support domain practitioners to engage in further domain construction without the presence of system developers. Second, the programming substrate only partially supported the rich bandwidth of communication necessary for collaboration and mutual learning between system developers and domain practitioners. The substrate supported mutual learning by enabling design stakeholders to ground their discussions in successive prototypes of the language. However, all this rich discussion was lost in that it was never recorded in a systematic way. Furthermore, on occasions when system developers were not present, practitioners needed to

remember any insights gained during prototype use or record them on paper for later communication to system developers.

SUPPORTING COLLABORATIVE DOMAIN CONSTRUCTION

In the first part of the case study, we saw that domains are socially constructed over time as domain practitioners try to improve their work processes. The resulting well-defined design languages improved their communication with other stakeholders. However, work practices, such as iterative design, were negatively impacted because domain practitioners were unable to make the next step towards computationally interpretable design languages.

The second part of the case study showed that developers can help practitioners to make this step and can also assist practitioners to envision new ways of working made possible by computational support. The VDDE story indicated how successive prototypes served as important artifacts for grounding communication and facilitating mutual learning between developers and domain practitioners. The story also showed that computational support is required for embedding this communication in the context of usage breakdowns. Finally, the story showed that computational architectures are required for enabling design languages to progressively evolve from ill-defined to well-defined to computationally interpretable.

In the remainder of this paper, we present an innovative tool-supported software development methodology enabling developer-practitioner communication and mutual learning.

Seeding, Evolutionary Growth, and Reseeding

To support domain construction, software development methodologies must explicitly acknowledge the collaborative nature of system design and that ongoing system adaptations to reflect changing work practices are inevitable. The SER development model [12] does this by viewing the system design process as consisting of three phases: seeding, evolutionary growth, and reseeding.

During the seeding phase, system requirements are not so much analytically specified as they are collaboratively evolved through an iterative, mutual learning process involving domain practitioners and system developers. This practice is what happens whether development methods acknowledge it or not [2, 5]. The SER model calls for requirements to evolve and relies on tools to make this process more effective. Particularly, tools supporting seed development assist domain practitioners in envisioning new ways of working and assist system developers in understanding current practices. The EVA system described next provides four types of design representation to facilitate this process.

The result of the seeding process is a seed (Figure 4) that has enough functionality for domain practitioners to use it for their normal work practices. Additionally, this seed must either provide computationally interpretable design languages or minimally provide the potential to be enriched with these languages during the evolutionary growth phase. Providing this potential requires system architectures that support domain objects and information to incrementally evolve through representations with increasing formality [31]; i.e. that support a seamless transition from ill-defined through well-defined to computationally interpretable design languages.

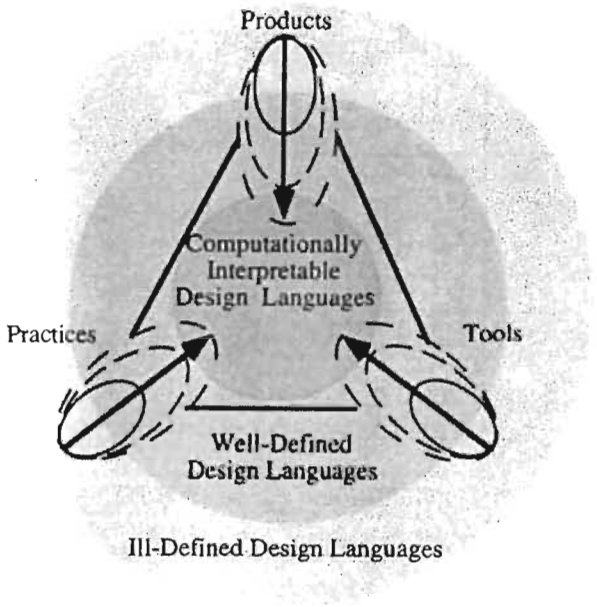


Figure 4. A Seed Supporting Domain Construction.

A seed is a system that has two properties: (1) it is used for normal work practices, and (2) it has the potential to be enriched to support computationally interpretable design languages that tightly couple tools, products, and practices.

During the evolutionary growth phase, domain practitioners modify the seed to better support changing work practices. These modifications can take the form of adding new information into the system or refining existing information or representations. However, practitioners typically do not have the programming knowledge to make the necessary modifications nor are they interested in programming per se [21]. We have prototyped innovative tools supporting domain practitioners to evolve their domain objects [31], the tools themselves [7], and underlying knowledge bases [14]. The incremental formalization approach [31] is particularly promising because it supports practitioners to progressively evolve informal textual communication such as electronic mail and simple text annotations embedded in their design artifacts to formal computational object-oriented representations. One challenge to this approach is where does this informal communication embedded in design artifacts come from? How does it get embedded into the design artifact? The Snippets and Expectation Agent systems discussed later were developed to address these needs. These systems enable

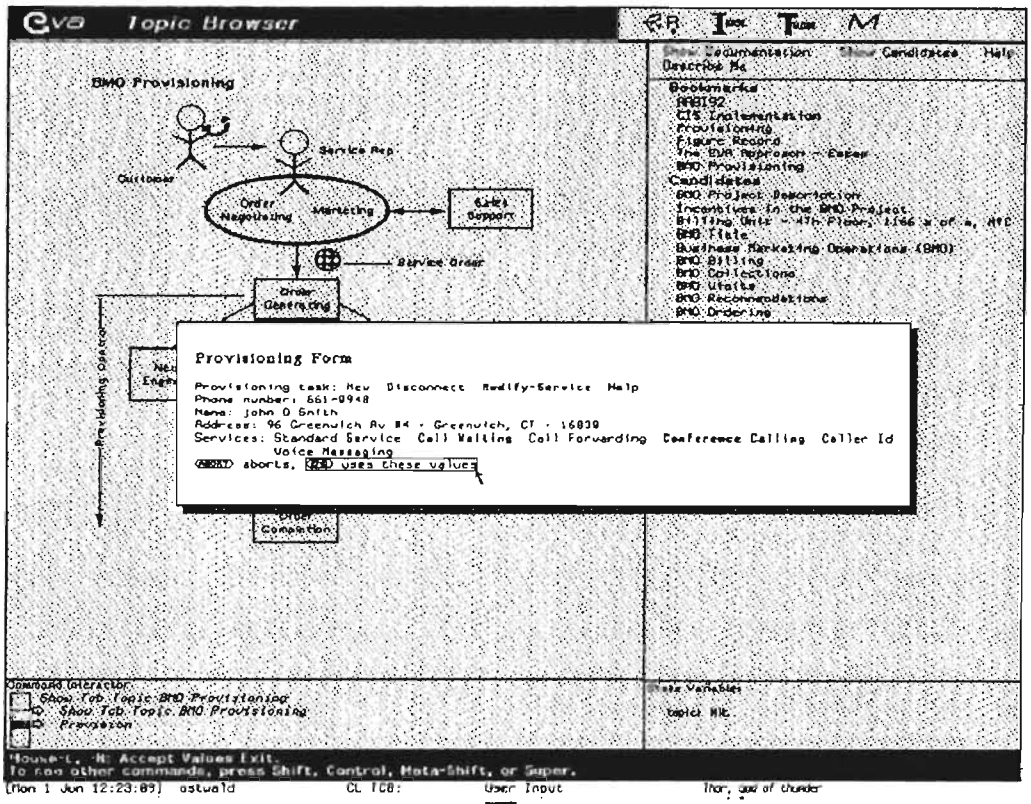


Figure 5: An Embedded Prototype in EVA

Here, the user has activated a prototype ("provisioning form" in white) by selecting an icon in a rich picture ("BMO provisioning" in gray). When the user is finished interacting with the prototype, it disappears from the screen and the user is once again in the rich picture.

practitioners to annotate seeds during use and channel electronic communication on design issues through the evolving seed and deliver this communication to the relevant design stakeholders.

During reseeding, system developers help practitioners to reorganize and reformulate the information previously entered. Additionally, reseeding provides an opportunity to reflect on current practices and again consider new ways of working.

EVA: Supporting Collaborative Seeding through Mutual Learning

EVA is a hypermedia-based tool which supports the collaborative development of seeds by supplying a medium for communication and mutual learning between practitioners and developers. EVA is based on the assumption that communication for mutual learning is best supported by external representations which are understandable by both practitioners and developers. Communication through design representations is difficult when the communication partners come from different workplace cultures. Design representations should be grounded in the current work practices, and yet support stakeholders to envision possible new domains. In the following paragraphs we describe four types of representations that EVA provides to support domain construction.

Informal Text and Graphics. Developers need to understand the current domain. A general understanding can be acquired through observation of users, interviews with users and managers, early design meetings, and written documentation such as preliminary system requirements. Information produced through these activities is stored in the EVA system, and provides a context for subsequent domain construction.

Rich Pictures. Based on a general understanding of the current domain, rich pictures [4] are created by developers to identify particular aspects of the existing domain that are problematic or vaguely understood. Rich pictures are ad hoc drawings or diagrams that serve as vehicles to help users explain their domain to developers. Rich pictures do not have a formal syntax, but do make use of symbols and diagrammatic conventions to represent a particular situation in a manner that is understandable by users. Rich pictures give users the opportunity to identify important aspects of their work, missing elements and incorrect terminology. Additionally, rich pictures serve to identify and understand well-defined aspects of the current domain.

Scenarios. While rich pictures help stakeholders to build a shared understanding of the current domain, scenarios help to build a shared vision of what the new domain should be. Scenarios in EVA are textual and graphic representations that describe tasks that users should be able to perform using a new system. Scenarios are expressed in domain language, using terminology and concepts made explicit in rich pictures. Because they are task-based, scenarios allow users to think about what they would like to do with a new system, rather than to articulate system requirements in an abstract context.

Prototypes. The emphasis of traditional scenario approaches is to help system builders understand the end user's requirements. In these approaches, scenarios help to uncover hidden implications and ambiguities in the requirements document. In EVA, scenarios provide a context for interactive prototypes, which are concrete representations that support stakeholders to understand and experience how formal design languages can support them to do envisioned tasks. Prototypes are vital for supporting domain construction because they let practitioners directly experience possible new ways of working.

In EVA, representations of the types described above are integrated into a single hypertext structure, thereby embedding prototypes into the documentation (see Figure 5). Practitioners access and interact with prototypes to experience what the final system might be like, and record their comments and critiques in the hypertext. The system design process is thus driven by communication between developers and end-users, and grounded by the evolving software artifact.

The EVA system was used to develop a prototype for service representatives at NYNEX. The service representatives were moving from a mainframe environment to desktop workstations and needed help envisioning how their work practices could be changed. The project was successful in promoting mutual education among design stakeholders in that system developers and service representatives were able to create a shared vision of what the new environment should be. Rich pictures proved effective for eliciting domain knowledge from practitioners and for grounding scenarios of future work practices. Practitioners and managers particularly appreciated that the progress of the design was clearly visible, a sharp contrast to traditional software development projects. This project's successful emphasis on communication inspired the creation of the next generation of software development tools described below.

Snippets and Expectation Agents: Supporting Communication during Design and Use

As mentioned above, in the process of designing a seed, a prototype is formed to explicitly describe the system being created. This prototype may be used to elicit and clarify requirements for the seed and in some cases, can evolve into the seed itself. The Snippets system was developed to keep track of requirements and design decisions, and support

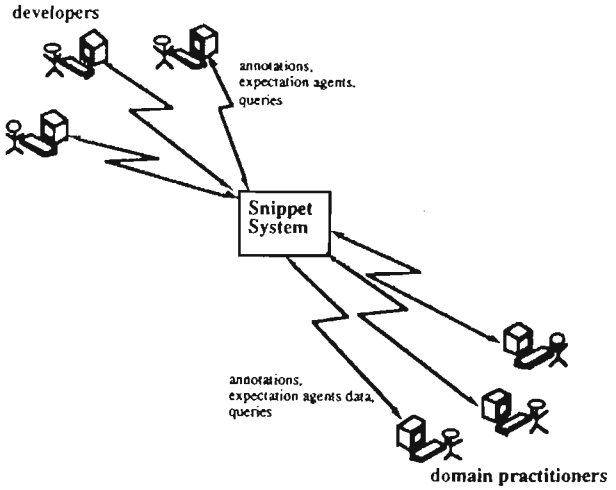


Figure 6: The Snippets system channels communications between design stakeholders.

domain practitioners and developers to communicate during system design and ongoing system adaptation. Snippets channels communication between stakeholders and stores it into a database (Figure 6). A database is formed containing this interaction history. This database can then be either browsed or queried.

There are two interfaces to the Snippets database: Annotations and Expectation Agents. Annotations are a passive front end to Snippets that allows practitioners to communicate information about the prototype to developers. Expectation Agents are an active front end to Snippets that prompts practitioners for comments if unexpected usage patterns are detected [15].

Annotations enable domain practitioners to record informal comments about the evolving prototype. These annotations are attached to the objects in the prototype and thus can be viewed by developers in the context of the use situation that triggered the comment. When an annotation is made, the Snippets system automatically forwards this information to the appropriate developers.

Expectation agents represent developers' expectation of how the seed will be used. In the design phase, domain

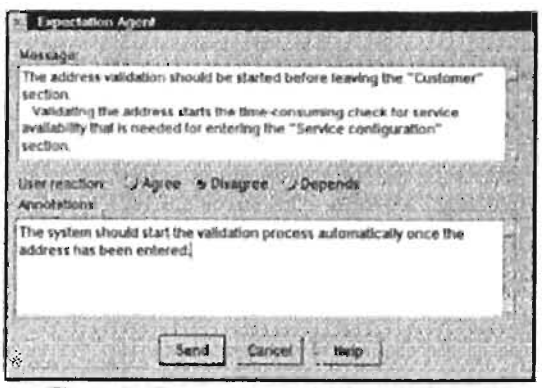


Figure 7: Expectation agent interaction.

practitioners often assess the prototype against scenarios of their work practices. If the prototype is used in a manner different from how the developer envisioned, this may indicate a breakdown in the shared understanding of what the system requirements are. Expectation agents catch this potential breakdown and promote discussion between the developer and the current user. This is done by initiating a dialog explaining the developer's expectation and eliciting a response from the user (see Figure 7). The user can then explain how this current scenario or set of scenarios deviate from the developer's expectation. This user response, along with an interaction history, is then forwarded to the developers as well as logged into the Snippets database. This active elicitation of user input promotes a larger group of domain practitioners to participate in the software design process.

While developing a system for service representatives at NYNEX, we observed how informal communication takes place. Users and developers pioneered practices, such as special notebooks dedicated to comments about the evolving prototype, that served to keep this informal communication tied to the artifact being developed. These observations indicate that it is beneficial to channel communication through the prototype since this communication is crucial for understanding how and why the prototype evolved [1, 16]. The snippets system capitalizes on the success of electronic mail as an informal communication device. By allowing communication to take place through the prototype, the information in the snippet database can then be progressively formalized into computationally interpretable design languages.

DISCUSSION

EVA, Snippets, and Expectation Agents have been applied in system building efforts inside NYNEX. EVA was applied successfully in the early development of a system for service representatives. It enhanced the process of mutual learning between system developers and service representatives through the use of embedded texts, graphics, rich pictures, scenarios, and prototypes. This collection of documents and representations was also used successfully to inform new developers joining the project. The Snippets system and the Expectation Agents have so far only been used in experimental settings. But informal evaluations and previous experiences with similar paper based methods suggest that such computational support is helpful.

EVA, Snippets, and Expectation Agents support communication between system developers and domain practitioners. As such, they enable a collaborative domain construction process that transcends the traditional domain modeling approach by dealing with the realities of fluctuating system requirements, changing domains, and evolving work practices. However in many contexts, organizational barriers have to be overcome before such an evolutionary development approach will receive the necessary support from management.

Another issue which has to be addressed is that the tools described in this paper are not sufficiently integrated. Currently, the information in EVA can not be accessed after the seed building process is completed. It would be more appropriate to make this information also available during seed evolution. This would not only allow stakeholders to discuss elements of the evolving seed but also allow them to refer to initial design and analysis information.

Another important issue we have not dealt with in the presented systems is that domain construction requires tools to support ongoing system development and system adaptation. Practitioners need support to evolve their design language and to extend the seed appropriately. In our research group at the University of Colorado at Boulder we have also built systems which support parts of this process (e.g. HOS [31], Modifier [14], and Programmable Design Environments [7]). So far these systems assume unchanging visual design representations for the evolving domain abstractions. The voice dialog design case study suggests that practitioners also need support in creating and evolving their own visual design representations in order to communicate with different design stakeholders. We are currently integrating and extending these tools to fit the domain construction model.

In addition to the communication-oriented tools we have presented here, system developers need computational support to implement the initial seed. One solution is to supply developers with a seed development substrate which already provides building blocks for basic components in a domain oriented system (e.g. construction kits and catalogs). In our research we have explored different architectures for such substrates (e.g. Agentsheets [26], HYDRA [13]). Further work is needed to turn these substrates into appropriate tools for domain construction.

SUMMARY

In this paper, we introduced an innovative software development model promoting domain construction and presented tools that support domain practitioners and software developers in this collaborative effort. The case study of the voice dialog design domain illustrated how social domain construction takes place in real world settings and enabled us to identify ways in which computational tools can support this domain construction process. Three tools - EVA, Snippets, and Expectation Agents - were presented that support seed development and evolutionary growth, particularly in the area of enhancing developer-practitioner communication. We are convinced that the combination of integrated communication tools such as these together with programming support tools for developers (e.g., substrates) and practitioners (e.g., incremental formalization) will enable the construction of domain oriented systems that are both useful *and* usable.

ACKNOWLEDGMENTS

We thank the designers at U S WEST Advanced Technologies and the service representatives at NYNEX for their help during this project. We thank Andreas

Girgensohn for his work with Expectation Agents. We also thank the members of the Center for LifeLong Learning and Design at the University of Colorado for helpful discussions on these issues. This research was supported by NYNEX, U S WEST Advanced Technologies, ARPA under grant No. N66001-94-C-6038, and NSF and ARPA under cooperative agreement No. CDA-940860.

REFERENCES

1. Atwood, M., B. Burns, A. Girgensohn, A. Lee, T. Turner and B. Zimmerman, "Prototyping Considered Dangerous," *To appear in: Fifth International Conference on Human-Computer Interaction (Interact '95)*, Lillehammer, Norway (June 27-29), 1995.

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

3. Byrd, T., K. Cossick and R. Zmud, "A Synthesis of Research on Requirements Analysis and Knowledge Acquisition Techniques," *MIS Quarterly,* March, pp. 117-138, 1992.

4. Checkland, P., *Systems Thinking, Systems Practice,* John Wiley and Sons, New York, 1981.

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

6. Ehn, P., *Work-Oriented Design of Computer Artifacts,* arbetslivscentrum, Stockholm, 1989.

7. Eisenberg, M. and G. Fischer, "Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 431-437.

8. Fischer, G., "Communication Requirements for Cooperative Problem Solving Systems," *Informations Systems,* Vol. 15, pp. 21-36, 1990.

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

10. Fischer, G., A. Lemke, T. Mastaglio and A. Morch, "Using Critics to Empower Users," *CHI '90*, Seattle, WA, 1990, pp. 337-347.

11. Fischer, G. and A. C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *HCI,* Vol. 3, pp. 179-222, 1988.

12. Fischer, G., R. McCall, J. Ostwald, B. Reeves and F. Shipman, "Seeding, Evolutionary Growth and Reseeding: Supporting the Incremental Development of Design Environments," *CHI '94*, Boston, MA, 1994, pp. 292-298.

13. Fischer, G., K. Nakakoji, J. Ostwald, G. Stahl and T. Sumner, "Embedding Critics in Design Environments," *The Knowledge Engineering Review,* Vol. 8, pp. 285-307, 1993.

14. Girgensohn, A., "End-User Modifiability in Knowledge-Based Design Environments," *Ph.D.,* CU-CS-595-92, Department of Computer Science, University of Colorado at Boulder, 1992.

15. Girgensohn, A., D. Redmiles and F. Shipman, "Agent-Based Support for Communication between Developers and Users in Software Design," *Knowledge-Based Software Engineering*, Monterey, CA, 1994, pp. 22-29.

16. Girgensohn, A., B. Zimmerman, A. Lee, B. Burns and M. Atwood, "Dyanmic Forms: An Enhanced Interaction Abstraction Based on Forms," *To appear in: Fifth International Conference on Human-Computer Interaction (Interact '95)*, Lillehammer, Norway (June 27-29), 1995.

17. Lewis, C. and G. M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?," *Empirical Studies of Programmers: Second Workshop*, 1987, pp. 248-263.

18. Nakakoji, K., "Increasing Shared Understanding of a Design Task Between Designers and Design Environments: The Role of a Specification Component," Department of Computer Science: University of Colorado at Boulder, Ph.D. dissertation, Dept. of Doctoral Dissertation (Technical Report: CU-CS-651-93), 1993.

19. Nakakoji, K., T. Sumner and B. Harstad, "Perspective-Based Critiquing: Helping Designers Cope with Conflicts among Design Intentions," *Artificial Intelligence in Design '94*, Lausanne, Switzerland (August), 1994, pp. 449-466.

20. Nardi, B., "Beyond Models and Metaphors: Visual Formalisms in User Interface Design," *Journal of Visual Languages and Computing,* Vol. 4, pp. 5-33, 1993.

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

22. Nardi, B. A. and J. A. Johnson, "User Preferences for Task-specific vs. Generic Application Software," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 392-398.

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

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

25. Reeves, B. and F. Shipman, "Supporting Communication between Designers with Artifact-Centered Evolving Information Spaces," *CSCW '92: Conference on Computer-Supported Cooperative Work*, Toronto, Canada, 1992, pp. 394-401.

26. Repenning, A., "Agentsheets: A Tool for Building Domain-Oriented, Dynamic, Visual Environments," Department of Computer Science (University of Colorado at Boulder), Ph.D. dissertation, Dept. of Ph.D. Dissertation, 1993.

27. Repenning, A., "Designing Domain-Oriented Visual End User Programming Environments," *submitted to: Journal of Interactive Learning Environments, Special Issue on End-User Environments*, 1994.

28. Repenning, A. and T. Sumner, "Programming as Problem Solving: A Participatory Theater Approach," *Workshop on Advanced Visual Interfaces (AVI '94)*, Bari, Italy (June 1-4), 1994, pp. 182-191.

29. Repenning, A. and T. Sumner, "Agentsheets: A Collaboration Medium for Creating Domain-Oriented Visual Languages," *To appear in: IEEE Computer (Special Issue on Visual Programming)*, 1995.

30. Rittel, H. and M. M. Webber, "Planning Problems are Wicked Problems," in *Developments in Design Methodology*, N. Cross, Ed., John Wiley & Sons, New York, 1984, pp. 135-144.

31. Shipman, F. M. and R. McCall, "Supporting Knowledge-Base Evolution with Incremental Formalization," *Human Factors in Computing Systems (CHI '94)*, Boston, MA (April 24-28), 1994, pp. 285-291.

32. Suchman, L., *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge University press, Cambridge, 1987.

33. Sumner, T. and M. Stolze (1995). Evolution, Not Revolution: PD in the Toolbelt Era. *To appear in: ProceedingsComputers in Context*, Aarhus, DK, (August 15-18) 1995.

34. Sumner, T., S. Davies, A. C. Lemke and P. G. Polson, "Iterative Design of a Voice Dialog Design Environment," *Technical Report*, CU-CS-546-91, Department of Computer Science, University of Colorado at Boulder, 1991.

35. Winograd, T. and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Addison-Wesley, Menlo Park, CA, 1986.