INTERACT '95 CONFERENCE PROCEEDINGS, pp. 179-184

Prototyping Considered Dangerous

Michael E. Atwood, Bart Burns, Andreas Girgensohn, Alison Lee, Thea Turner, and Beatrix Zimmermann

NYNEX Science and Technology
500 Westchester Avenue
White Plains, NY 10604 USA
{atwood, bart, andreasg, alee, thea, bz}@nynexst.com

KEY WORDS: prototypes, prototyping, scenarios, domain-oriented design environments, design intent, expectation agents.

ABSTRACT: In this paper, we argue that prototypes can hinder, rather than aid effective communication. The dangers are: (1) prototypes may contain hidden assumptions which might surface too late; (2) obtaining feedback in the context of use is prohibitively expensive and rarely done; and (3) partly as a consequence of these problems, prototypes cause a focus on displays and other surface features of computers, not on the more difficult problem of how people function in their environments to solve problems. We propose *design intent* which specifies how a system will fit in and interact with the environment in which it is placed and *expectation agents* which monitor the system in use and detect uses counter to the intent as ways to alleviate these dangers.

1 INTRODUCTION

The basic problem in large software development efforts is communication — not communication between modules or communication between hardware and software systems, but communication between people. Recognition of the need for better communication between the users of systems, the developers of systems, and other effected persons is a major motivation for the use of prototyping.

In this paper, we argue that prototypes, as frequently used, can hinder rather than aid effective communication. We will not categorically argue that prototyping efforts should cease. However, as with any tool, prototypes can be misused as well as used effectively. The seductive nature of quickly building a prototype artifact carries with it three alluring dangers. First, prototypes may hide, rather than highlight, some of the assumptions made during their development. Second, obtaining feedback in the context of use is prohibitively expensive and rarely done. Third, and, partly as a consequence of these problems, prototypes cause a focus on surface features such as displays instead of on the more difficult problem of how people function in their environments to solve problems.

We first describe these dangers and then describe our current approach to avoiding them.

2 THE COMMUNICATION PROBLEM

The importance of effective communication in the development process is highlighted in a detailed look at seventeen large software projects in nine companies (Curtis, Krasner, and Iscoe; 1988). Based on interviews with project team members, they noted three frequently recurring problems, all related to the lack of effective communication between those involved in the design effort: (1) the thin spread of application knowledge, (2) emerging, fluctuating and conflicting requirements, and (3) communication bottlenecks and breakdowns.

The first problem, gaining application knowledge, acknowledges that application knowledge is a crucial part of design. As one participant in the Curtis, Krasner, and Iscoe study noted, "writing code isn't the problem, understanding the problem is the problem."

Even on seemingly simple applications, all the necessary application knowledge may be spread over dozens of individuals. Omitting any of this knowledge, through a lack of communication or simple oversight, jeopardizes a successful development effort. To avoid

the "Tower of Babel" problem (Brooks, 1975), communication between diverse groups of people is required.

The second problem, fluctuating and conflicting requirements, arises primarily because computer systems fit into larger environments which are not themselves static. While requirements sometimes change because people involved in the development effort rethink earlier decisions, more typically, they change to reflect the fact the environments are changed. Because the environments into which systems are to be placed continue to evolve, problem understanding must proceed continuously and simultaneously with other system development activities. New requirements are likely to emerge as understanding matures.

Similarly, the third common problem, communication bottlenecks and breakdowns, does not typically occur because one group neglects to communicate with another. Rather, they occur because one group is not able to **effectively** communicate with another. The underlying problem is that many people are required to develop a system, but each has their own view of the system, does not recognize how many others have different views of the system, and cannot communicate effectively with many of the others. Our solution to this "symmetry of ignorance" problem (Rittel, 1984) is to provide a common artifact through which many people can communicate. While this is the solution that prototyping intends to implement, we will argue later that it can often fail to do so.

3 FLAWED IMPLEMENTATION OF NOBLE INTENT

A prototype is a common artifact that represents the evolving system under development. This is exactly as it should be! What, then, are the dangers that prevent us from achieving this goal? We consider three dangers to be most serious.

3.1 Assumptions may be hidden

Prototypes are considered to be useful to all concerned with a development effort in facilitating a convergence on the system's requirements. Requirements specify *what* the system must do and should result in specifications that specify *how* the system will function. What prototypes are most in danger of losing is the *why* information that records the assumptions underlying these decisions.

For example, consider a hypothetical system that allows customers to call a representative to order new telephone services. *What* the system should do is clear — it will answer a telephone call, connect the customer to a representative, determine what services are available to the customer, and support the order entry process. *How* is equally clear — we will use a commercial answering system to take calls and route them to a representative, write a program that queries an existing system to determine what services are available, and then uses a form-based interface to support order entry. A prototype of this system should take hours, at most, to develop. (Berghel's (1994) comments on the dangers of "vacuous prototyping" are worth noting here. He clearly notes the dangers of prototyping what cannot be built.)

In constructing our prototype, we made several assumptions. We assumed a certain call arrival rate when selecting the answering system; we assumed a certain response time with an existing data base when we designed the interface; we assumed that orders would be completed in one transaction when designing the data base for customer records. Failure to record these assumptions — the *why* information — will complicate our efforts to refine this prototype.

During tests, we found that the call rate and number of customers was more than expected, that the response time with the database can occasionally take minutes, rather than seconds, and that customers often must make multiple calls in order to gather all the information needed for an order. These are exactly the problems that we later see as communication breakdowns ("why didn't you tell me how many customers would call at once?"), fluctuating requirements ("you mean we have to store and retrieve partial orders?"), and a thin spread of application knowledge ("who should have known that data base access would be so slow?"). All very real problems; all created by failing to note carefully *why* information.

3.2 Feedback may be too expensive

Failing to record *why* information also makes analyzing a system more difficult. The usefulness and usability of a system can only be measured with respect to actual users performing actual tasks in their actual environment on working systems (cf., Gray, John & Atwood, 1993; Atwood, Gray & John, in press; Turner, Lee, & Atwood, in press). We do not claim that obtaining feedback is too expensive because systems must be substantially developed before they can be evaluated. This is clearly true, but we make a stronger argument. Feedback is too expensive because we forget to record the assumptions that would let us link analysis results with design decisions.

Returning to the order entry system discussed above, assume that we are in a field trial. The most significant point that comes from analysis is that too many customers hang up the telephone before completing a transaction. This problem must be corrected if the system is to be successfully used. What aspect of the system do you change to correct this problem? Do people hang up because the initial prompt is in a language they do not understand, because the response time to query the data base is too long, because the representative asks for information they view as inappropriate, or what?

The common approach to analysis is to measure all that can be measured and then make inferences about what is happening. Only a small fraction of this data is useful, however. We made assumptions about how the system would be used. That the system is not used successfully indicates that one or more of these assumptions is not met in practice. Did we assume that most speakers would understand the initial natural language prompt, that the data base query would take no more than one second, or what?

Analysis only indirectly indicates that the system must be changed. Directly, analysis indicates that our assumptions about system use must change. Because we do not typically record these assumptions, however, we struggle to relate analysis results to properties of the system, rather than simply relate them to the assumptions underlying these properties.

3.3 Inappropriate human-computer interaction issues are highlighted

As a community, we often identify ourselves as *human-computer interaction* specialists. This is an inappropriate attribution and one that denigrates our contributions to system development. It leads to focusing inappropriately on the artifact that is the tool through which we do our work rather than on the primary focus of our work.

Prototyping can cause a clear focus on the artifact rather than on the use of the artifact. With the prototype discussed above, we contend that a prototype is more likely to cause discussion of whether the initial prompt should say “hello” or “welcome” than on whether the initial prompt should convey that multiple natural languages are available. The problem, as noted by Ramamoorthy, Prakash, Tsai, and Usuda, 1984, is that prototypes can cause a focus on nonfunctional, rather than functional, requirements The prototype artifact highlights nonfunctional aspects and obscures the reasons underlying the functional decisions.

4 RELATED RESEARCH

Recognition of the need to capture the *why* information is behind the current interest in *design rationale*. Design rationale captures the history of the arguments that are associated with design decisions. This information is frequently structured formally; in IBIS (Rittel, 1984) and gIBIS (Conklin & Yakemovic, 1991) it is structured in terms of *issues, answers, and arguments*, while in QOC (McLean, Young, Bellotti, & Moran, 1991) it is structured as *questions, options, and criteria*. In DRL (Lee & Lai, 1991), the structures are equally formal but much more elaborate. The focus of these approaches is both on *what* the system is and *why* it is the way it is.

However, while many currently claim that design rationale will have value, there are few studies suggesting any value. Of the six papers in one issue of the journal *Human-Computer Interaction* devoted to the issue of design rationale, only one (Conklin & Yakemovic, 1991) described the application of design rationale to a real project and presented anecdotal evidence that it was of value.

Why is an idea that has such broad appeal on both theoretical and intuitive grounds used so little? We believe there are two reasons. First, they violate the “benefit-work rule” that governs collaborative efforts. That is, benefits should go to those who do the work. With existing schemes, however, designers do the work, but others benefit; as a result, this work is not done well or not done at all. Second, existing schemes focus on what designers know; this may include application knowledge that they have abstracted from others, but it does not directly involve all those who directly represent this application knowledge. We argue that constructing this information as it evolves increases effective communication among all people involved and that this increase in effective communication is of sufficient benefit that this process will be followed.

5 COMMUNICATION, COLLABORATION, AND CONTEXT — AVOIDING THE DANGERS OF PROTOTYPING

We propose an evolutionary model of software development in which the expected utility of a system is improved by active, computer-based support for (1) feedback from users of prototype systems to developers, (2) communication and collaboration among us-

ers and developers, and (3) mutual education among user and developer communities.

Our approach is based on real development experiences and current research on agent-based software environments. Previous projects — Ernestine (Gray, et al; 1993; Atwood, et al, in press), New Horizons/OSDI (Turner, Lee, & Atwood, in press), and Bridget (Atwood, Burns, Girgensohn, & Zimmermann, 1994) — highlighted for us the importance of feedback, communication, and mutual education in system development efforts. This realization also caused us to work on methods to actively support these activities.

Project Ernestine provided us with critical insight: Design of systems to function in complex situations, such as large technology-oriented companies or interdisciplinary design domains, requires a deep understanding not only of the application domain, but also of the practice (Bjerknes et al, 1985) of the people who will use the systems. In order for useful systems to be built in these conditions, system design must be based on a process of mutual education (Ehn, 1988) between system builders and users.

The goal of mutual education is to allow all members of the design team to contribute their expertise to the system building process. The most important resource in the system building process is an artifact that all stakeholders can understand and contribute to and which serves to build a sense of community among the stakeholders. We call this resource *design intent*.

6 DESIGN INTENT

Design intent specifies how the system will fit in and interact with the environment into which it will be placed. It includes design decisions, changes to decisions, and the rationale for those decisions and changes. While a focus on interactions with the environment, rather than just a description of the system's own operations, is not a common practice in software development, we believe it is an appropriate practice, because of the importance of understanding the environment (cf., Gray et al, 1993; Atwood et al, in press, Turner et al, in press).

Design intent derives from communications among the individuals and groups associated with a development effort and becomes the focus for communications among these individuals and groups (see Figure 1). There is no generic or canonical representation of design intent. Unlike the more prescriptive notations of design rationale, design intent is more process than tool. The key idea is to facilitate and capture the communication among the participants in a design effort. The form and content of design intent are determined more by the problem addressed and the people involved than by proscriptive ideas about notational format.

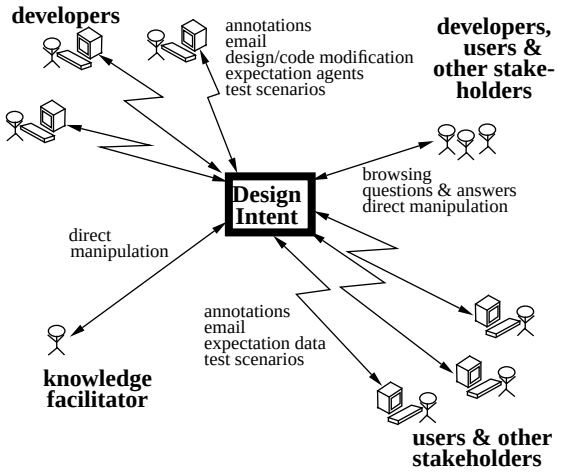

Figure 1: Design Intent Scenario

The initial steps in forming design intent are common with many other approaches — we talk to those involved, listen to what they have to say, and reflect our understanding back to them. A difference is that we structure this mutual understanding as it evolves. To illustrate the various ways in which design intent is used to facilitate communication, we will step through a brief scenario.

In this scenario, a customer service representative (CSR) is interacting with a customer who has called to order new telephone services. In order to do this, the CSR must determine whether the customer has existing service, what their credit rating is, how to contact them later, and the address where the services are to be installed. The address information is important for two reasons. First, the CSR needs to verify that the address is one that actually exists so that the installer can find it. Second, the address is associated with a wire center and it is the wire center and the equipment it contains that determine what services are available.

In observing the CSRs interactions with customers, the system developers note that the CSRs typically validate the address early in the dialogue but query what services are available at that address much later

in the dialogue. Since the query for available services may take minutes, this can increase the total interaction time. The system developers, therefore, propose that the query for available services be automatically initiated by the system once the address is verified and that addresses be verified before initiating other aspects of the transaction, such as collecting billing information. This proposal is accepted by the CSRs working with the developers as one that saves them the effort of manually performing this operation and that saves time. The understanding between the developers and the CSRs is represented as design intent.

The understandings encoded in design intent are coded by the developers as expectation agents (Girgensohn, Redmiles, & Shipman, 1994). Expectation agents actively monitor the system in use to detect uses that are counter to the intent. While expectation agents are initiated by the system and comment on how the user is interacting with the system, annotations are initiated by the user and comment on the system performance from the user's perspective. The value is in helping to understand how the system is being used, where it needs to be improved, and especially where the developers (or others) have "bugs" in their understanding of the system requirements.

Expectation agents are active and can initiate dialogues with system users at any point where they are instructed to do so, either immediately when triggered, at the end of the current interaction with the customer, or at another time. This mode of operation is influenced by Schön's (1983) view that professionals learn during "breakdowns" — situations in which well-learned procedures fail to work. Either users learn a more effective mode of operation or developers learn where requirements need to be refined. Again, the focus on having an active, rather than passive, understanding of the system is not a common practice, but, we believe it should be.

In this scenario, the design intent specifies that address validation should be started while the CSR and customer are in the "customer" section of the order entry form. However, the CSR leaves this section before validating the address in order to make an entry in the "billing" section. This triggers an expectation agent (see Figure 2). The CSR may then comment on this agent. In the case shown in Figure 2, the CSR disagrees with the agent and the underlying design intent and indicates why. Because the agent is tied to the underlying code and the system maintains a list of who is responsible for each section of code, the CSR's reply is directed to the appropriate developer.

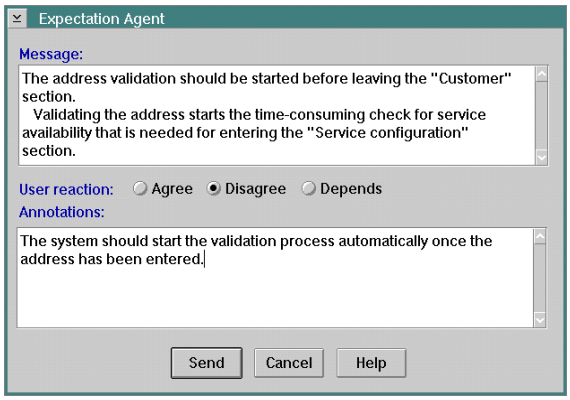

Figure 2: Expectation Agent Interaction

The interactions between developers and users, and others, centered around expectation agents are effective ways to refine their mutual understanding of the system requirements and the environment in which the system will sit. We contend that situating this requirements refinement process in the actual work setting facilitates both the elicitation of requirements and the validation of those requirements.

Annotations are similar to expectation agents in that they allow for comment on expected performance. They allow the user to initiate a dialogue on system performance, as opposed to allowing the developer to initiate a dialogue on user performance through an expectation agent. Since the system maintains a list of who is responsible for what section of code, the annotations are directed to the responsible developer, as well as being recorded in the design intent document.

Annotations and responses to expectation agents may be text or multi-media. If a user prefers, for example, an audio reply or an audio-video reply can be sent to the developers. The history of the interaction that preceded the breakdown can also be included to give the developer additional context information. All this information is associated with some aspect of the design intent documentation and all is directed to an individual or an identified group of individuals. That is, information relevant to a development effort is not allowed to "float" with no relation to other information nor is information sent to a "general delivery" address where the sender is not aware of the identity of the recipient.

7 SUMMARY

The basic problem with any large scale development effort is communication. During a development ef-

fort, a great deal of information is generated and exchanged. However, it is apparent that all of the information necessary to the development process is not communicated effectively. Some necessary information is not generated or not exchanged appropriately; some information is not maintained throughout the development effort, preventing its communication to the others that may need this information later in the development effort.

Design intent provides solutions to the communication problems associated with large scale development efforts. We avoid the problem of the *thin spread of application knowledge*, by capturing domain knowledge in a form that can be shared with others; we mitigate the problem of e*merging, fluctuating and conflicting requirements* by maintaining a the history of design changes and the rationale underlying those changes; we reduce problems of *communication bottlenecks and breakdowns* by ensuring that all communications are related to the design intent documents (they do not "float free") and that all communications are addressed to individuals (they do not go to "general delivery"). Most importantly, it develops a sense of community among all those involved in the development effort.

REFERENCES

Atwood, M.E., Burns, B., Girgensohn, A., & Zimmermann, B. Dynamic forms: Intelligent interfaces to support customer interactions. Technical Report. White Plains, NY: NYNEX Science & Technology, 1994.

Atwood, M.E., Gray, W.D., & John, B.E. Project Ernestine: Analytic and empirical methods applied to a real-world CHI problem. In P.G. Polson & C. Lewis (Eds.) Success Cases, Emerging Methods, and Pragmatic Context in Human-Computer Interaction. San Mateo, CA: Morgan-Kaufman, in press.

Berghel, H. New wave prototyping: Use and abuse of vacuous prototyping. interactions, 1(2), 1994, 49-54.

G. Bjerknes, T. Bratteteig, J. Kaasboll, K. Nygaard, I. Sannes, H. Sinding-Larsen, G. Thingelstad. Gjensidig Laerning: Florence Rapport fra fase 1 [Mutual Learning: Florence Report No. 1]. Department of Informatics, University of Oslo, Norway, 1985

Brooks, F.P., Jr. *The mythical man-month: Essays on software engineering.* Reading, MA: Addison-Wesley, 1975.

Conklin, E.J. & Yakemovic, K.C.B. A process-oriented approach to design rationale. *Human-Computer Interaction*, *6*, 1991, 357-391.

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

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

Girgensohn, A., Redmiles, D., & Shipman, F. Agent-based support for communication between developers and users in software design. In Proceedings of the 9th Knowledge-based Software Engineering Conference, Monterey, CA: IEEE Computer Society. 1994

Gray, W.D., John, B.E., & Atwood, M.E. Project Ernestine: Validating GOMS for predicting and explaining real-world task performance. Human-Computer Interaction, *8*, 1993, 237-309.

Lee, J. & Lai, K.-Y. What's in design rationale? *Human-Computer Interaction*, *6*, 1991, 251-280.

McLean, A., Young, R.M., Bellotti, V.M.E., & Moran, T.P. Questions, options and criteria: Elements of design space analysis. Human-Computer Interaction, *6*, 1991, 201-250.

Ramamoorthy, C.V., Prakash, A., Tsai, W-K., & Usuda, Y. Software engineering: Problems and perspectives. IEEE Computer, October 1984, 191-209.

Rittel, H. Second-Generation Design Methods. In N. Cross (Eds.), Developments in Design Methodology (pp. 317-327). New York: John Wiley & Sons, 1984.

Schön, D.A. The reflective practitioner: How professionals think in action. Basic Books: New York, 1983.

Turner, T., Lee, A., & Atwood, M.E. (in press) Usability engineering: Do we practice what we preach? In: Nielsen, J. (Ed.), Advances in Human-Computer Interaction vol. 5, Ablex, Norwood, NJ, in press.