In T. Moran & J. Carroll (Eds.), *Design Rationale: concepts, techniques, and use*, 453-470. Erlbaum, 1996.

Evaluating Opportunities for Design Capture

Jonathan Grudin

Interviewer: What percentage of development projects are terminated before they're completed or before the product gets out the door?

Successful development manager at a major software company: Ninety percent.

Interviewer: (laughs) Seriously, what would you estimate?

Development manager: Ninety percent.

ABSTRACT

A key motive for developing ways to structure and preserve records of decision-making in software development is the belief that the high cost of system maintenance could be reduced by providing access to these records. Development projects vary widely. Tools and methods can be appropriate for one development environment but not for another. In this chapter, I describe four kinds of software development environments and outline the opportunities and obstacles for using design rationale in each. Some development projects have good reason to avoid investing resources early in development, even when great benefits may accrue later. Many off-the-shelf product development efforts are arguably of this nature. In other projects, such as customized software development, the value of upstream investment seems more compelling. I reflect on the difficulty of learning from failed projects and on the need for researchers to distinguish carefully between the requirements of science and those of engineering.

1. INTRODUCTION

The recording of design rationale is an investment of resources early in the development process ("upstream"). It is expected to provide significant savings downstream during design change, enhancement, and other forms of maintenance. Maintenance is often cited as consuming up to 60% or even 90% of total software development costs (e.g., Boehm, 1981; CSTB, 1990; Fischer et al., 1992; Jarczyk, Loffler, & Shipman, 1992; MacLean et al. (1991 [Chapter 3 in this book]). This provides a powerful argument for upstream investment, but a crucial datum is always overlooked: Many development projects have little or no downstream activity that will benefit from such an investment—*because they are not completed*. Even when a project is completed, a different organization is often responsible for maintenance, in which case an upstream investment would not benefit the development organization. And any project, by diverting resources to capture design rationale, may reduce its likelihood of surviving or succeeding. The challenge for those who promote design capture is thus twofold: First, to

identify projects that have a relatively high probability of surviving long enough to require maintenance; then, to assure that a favorable balance of costs and benefits is realized by those participants who contribute to recording design rationale.

Consider the situation described in the opening interview. Assume that the manager is correct and assume that nothing is preserved when a project is terminated (these assumptions are examined later). Ninety percent of the time, investment in recording design rationale for the benefit of corrective, adaptive, perfective, or preventive maintenance is lost outright due to project termination. Furthermore, a project that has a chance to survive, that could join the successful 10%, might, by diverting resources to record design rationale, slow its progress, fail to reach a critical checkpoint, and be terminated. When the upstream waters are infested with piranhas, your best strategy can be to get downstream as quickly as possible, even if you arrive less than fully prepared for all possible contingencies. (We will meet some piranhas later.)

In commercial product development, project completion rates are low. (The case study by Sharrock and Anderson (1994 [Chapter 15 in this book]) is a nice description of a terminated project.) But in other development contexts, they are higher. For example, expensive systems built under contract are usually delivered. All else equal, a tool to capture design rationale for facilitating maintenance will be more valuable where project completion rates are high. However, completion rate is only one factor in assessing the appropriateness of design capture. Another factor is the continuity of personnel. Maintenance may be handled by the original developers, by different groups within the same organization, or be the responsibility of an independent organization. Clearly, this also affects the value to the developers and the development organization of an upstream investment to facilitate subsequent maintenance.

The message for those developing systems and methodologies to support the recording of design rationale is this: Give careful thought to your prospective users. Many design rationale efforts are directed toward an audience that may not offer the best conditions for success, the developers of commercial off-the-shelf software. Design rationale research will be more successfully applied if it is directed to appropriate audiences.

Of course, one can always work to alter adverse conditions. Organizational change could improve the conditions for design capture—clear away the upstream piranhas. But organizations change slowly. Someone is fond of the piranhas or they would not be there, and design capture does not yet have a track record that gives weight to arguments for major change.

Below, I first present the perspective on design rationale taken in this chapter. Then I suggest that many analyses have been limited by being restricted to the examination of successful projects. The utility of capturing design rationale is discussed for four development contexts: off-the-shelf product development, internal or in-house system development, large competitively bid development contracts, and smaller non-competitive consulting or third-party development

projects. Each context presents unique opportunities and obstacles for utilizing design capture techniques, but they are far from equal in promise. The possible value of information preserved in the course of unsuccessful projects is then considered. Finally, I examine the fit between approaches based on the methods of scientific research and the requirements of engineering and development practice.

2. THE USE OF DESIGN RATIONALE IN SYSTEMS DEVELOPMENT ORGANIZATIONS

In this section, I focus on considerations affecting the capture and use of design rationale in systems development organizations, primarily software development. This is a natural focus for the developers of software tools to support design capture, although Sharrock and Anderson (1994 [Chapter 15 in this book]) and Gruber and Russell (1994 [Chapter 11 in this book]) present examples from other design domains. Consideration of other design domains could only add to the variations in design environments and increase the case made in this chapter for considering the conditions that exist in a given setting for capturing rationale.

As noted by Conklin and Yakemovic (1991 [Chapter 14 in this book]), design rationale is captured and used routinely in development organizations. It is captured in the memories of individual developers, in meeting minutes recorded by secretaries, in dictaphone tapes and in memos. Written specifications may include some rationale. Development organizations sometimes preserve video records of design meetings and presentations. Engineering notebooks are often used conscientiously to record rationale. When working as a developer, I used all of these techniques at one time or another. Each had its uses and its weaknesses. Among the weaknesses are tendencies to be incomplete, unstructured, and/or dispersed throughout an organization; thus records are difficult to retrieve and interpret later, and can be impossible for third parties to access.

The current interest in this topic derives from the observation that technology could address these weaknesses. It can help with retrieval and access. By supporting formal or semiformal systems for representing information, technology might assist in building more interpretable and complete accounts of the design process. This chapter focuses on such computer-based systems, although some of the conclusions will apply to other forms of design recording.

What is the rationale for design rationale research? The most common reason given for capturing design rationale is to use it for subsequent consultation, for use in redesign or maintenance. Another reason is to produce improved designs through the discipline or focus of attention required to derive or capture design rationale (e.g., Carroll & Rosson, 1991 [Chapter 4 in this book]). Several authors mention both as potential benefits; for example, MacLean et al. (1991 [Chapter 3 in this book]) present an extensive case for the benefits of design rationale in system maintenance, but also speculate that designers might be rewarded through immediate design improvements. A third reason for recording rationale is for communicating progress to other project members.

This chapter focuses on the first rationale noted above: the benefits to subsequent development or maintenance. This has tremendous intuitive appeal. It is less clear that formal or semi-formal design rationale representations will be useful in communicating design progress; the little available evidence is negative (Conklin & Yakemovic, 1991 [Chapter 14 in this book]). Similarly, although outstanding researchers have improved designs when using these methods, trials with less brilliant users have not fared well (e.g., Fischer et al., 1992, pp. 402-403). One impressive design improvement occurred not when the design rationale was being recorded by developers, but when researchers translated it from one representation format to another (Conklin & Yakemovic, 1991 [Chapter 14 in this book]).

My analysis is comparative. Conklin and Yakemovic discuss the tradeoff between costs and payoffs for design capture. Some cost is assumed: If a design tool cost nothing to use, it could be used without harm in every development context. Similarly, a limited benefit is assumed: If a design tool provided virtually infinite benefit, it would be used in every development context. Existing tools lie in between: Mastering and using them requires additional effort on the part of developers, and it is hoped that a net benefit will result. The additional effort is concentrated, along with design activity, at the outset of a project, consistent with the evidence that design errors are less costly if caught early. My central question is: Under what conditions will the early, extra effort most likely be beneficial? Having answered that, we can dig further for insight into improving the tools.

The same analysis holds for tools or methods that improve design directly. The utility depends on the cost of the effort and the benefit it provides. If an immediate benefit outweighs the cost, it is a winner. If realization of the benefit is delayed, then more or less favorable conditions probably exist for applying it.

These cost/benefit analyses are one of three major considerations in applying design capture in specific settings. The other two are not addressed further in this chapter but must also be evaluated in specific development contexts:

1) Will social, political, and motivational concerns prevent the explicit statement of the real reasons underlying design choices. This can occur even in engineering settings. The rationale might contain strategic information of use to competitors; a certain group might not be deemed capable of a tricky implementation, but it is not politic to state this explicitly; the introduction of the design capture tool could shift decision-making power to skillful tool users, who could game the new system; fears of these things could affect tool adoption. And so on. Examples can be found in Sharrock and Anderson (1994 [Chapter 15 in this book]) and a discussion of some of these issues concludes Conklin and Yakemovic, 1991 [Chapter 14 in this book]).

2) Accurate interpretation of captured design rationale could require more knowledge of the context that existed at the time of capture than can possibly be recorded. This issue is addressed by Gruber and Russell (1994 [Chapter 11 in this book]).

3. "REVERSE-ENGINEERING" SUCCESSFUL PROJECTS: PROBLEMS WITH A SYSTEM PERSPECTIVE

As more large software programs are written and as they age with varying degrees of gracefulness, challenges in software maintenance become more pressing. Boehm (1988) argued for considering maintenance to be inseparable from development, for freeing it from its present "second class" status in software engineering. In "Scaling up: A research agenda for software engineering," the Computer Science Technology Board also focused on maintenance. They wrote "System maintenance may constitute up to 75% of a system's cost over its lifetime. The high cost of maintenance makes designing for a system's entire life cycle imperative, but such design is rarely if ever achieved." (CSTB, 1990, p. 282)

Given the apparent benefit to be gained through upstream design investment that anticipates maintenance needs, why is such design "rarely if ever achieved"? Why do we neglect an apparent opportunity to benefit economically? Why, as recounted by Conway (1968, p. 29), is there "never enough time to do something right, but always enough time to do it over"?

The answers are not hard to find. In the next section, obstacles to upstream investment in design are identified in each of several different systems development contexts. They have not been recognized because of assumptions that have biased analyses of systems development projects:

1) *Analyses are system-centered.* In considering the development process, our focus is on the system being developed. This diverts attention from the specific individuals involved in the process. Focusing on development phases reduces our awareness that different people are generally involved at different times. Focusing on the system being developed also obscures aspects of the organizations in which development occurs, including "tangential" organizational factors that can act to terminate projects prior to completion.

2) *Analyses are based on large systems.* The study of large systems, particularly in the context of government contracts, gave rise to the field of software engineering. The challenges and risks are greater for large projects; nevertheless, it is a mistake to apply data or models derived from their analysis to contexts in which smaller projects are the rule.

3) *Analyses of development projects are retrospective and aggregate.* This leads to a model of a complete development process, with no consideration given to early termination. One rarely sees descriptions of the waterfall model in which the water fails to fall to the bottom. Only recent variations such as Boehm's (1988) spiral model provide explicit provisions for exiting when a risk analysis produces an unfavorable assessment.

This idealized project perspective obscures the fact that many projects never incur maintenance costs because they are canceled. Statements such as "maintenance... can occupy as much as 90% of the effort in the software life cycle" (MacLean et al., 1991 [Chapter 3 in this book]) are clearly based on successfully completed projects. One does not read "in many contexts,

maintenance occupies 0% of a typical system's cost," although this is equally true. The system-centered perspective obscures the fact that design, development, and maintenance engineers are typically different individuals, which can reduce or eliminate the former's incentive to design for maintenance. Even when an incentive exists, lack of communication among the engineers reduces the awareness of how useful design rationale might be. Our attention to relatively large projects, often contract or internal development projects, neglects important development contexts in which different conditions prevail.

A corrective is to broaden our analytic focus, considering not individual projects but instead the organizational contexts in which development occurs. A decision to embrace design capture, like any approach to design and development, is taken on the basis of evolving conditions within a development organization. Examination of these conditions enables us to assess the obstacles and opportunities for innovation in upstream design activities.

Four organizational contexts for systems development will be examined: product development, internal development, competitively bid contract development, and customized software development. Although not an exhaustive set, they provide a range of opportunities for and obstacles to recording and using design rationale. Design capture tools or techniques may require adaptation for use in specific environments. They do not seem equally promising in all cases.

4. PROSPECTS FOR DESIGN CAPTURE IN SEVERAL DEVELOPMENT CONTEXTS

Substantial design rationale research is carried out by employees of computer and software vendor companies and by academic researchers sharing their concerns. In the late 1970s, large markets for word processors, spreadsheets, graphics and other interactive applications caught the attention of computer companies and spawned software development companies. In the early 1980s, conferences and journals concerned with creating usable software products appeared with names involving permutations of the words "human," "computer," and "interaction." Exploration of design rationale is found in these settings, much of it carried out by cognitive psychologists with an enduring interest in issues in problem-solving, memory, and other related activities. Much of the work in this book was first outlined in a workshop devoted to design rationale at a Computer and Human Interaction Conference (CHI'91) and published in the journal *Human-Computer Interaction*. This book's non-academic contributors are from Digital, IBM, Kodak, MCC, NCR, and Xerox—all commercial product developers and marketers.

Important as commercial, off-the-shelf product development is, more resources are devoted to in-house software development, carried out by information systems groups in hospitals, banks, insurance companies, government agencies, and so forth. Variously called data processing (DP), information systems (IS), management information systems (MIS), or information technology (IT), this community has conferences and journals that focus primarily on large systems that serve organizations.

A third influential development context is competitively-bid contract development. The U.S. government, the largest computer user, sponsored most of the major software development projects of the 1950s. As a result, many software engineering methods in use today evolved in the context of contract development. These large systems encounter significant maintenance issues and interest in design rationale is growing in this community.

Many smaller organizations that have recently computerized cannot afford full-time software developers, yet want software customized or adapted to their individual needs. A growing number of consulting and third-party development companies specialize in customized software development. Addressing specific vertical markets or application domains, these companies often seek to reuse software in moving from customer to customer. With less formal or restrictive procurement, the conditions surrounding development are different than for large competitively-bid contracts.

When we think in terms of "the computer industry" we overlook how very different these and other branches of systems development have become. Different issues, approaches, and challenges govern each (Grudin, 1991a). Developing and positioning methods and tools to support design argumentation require careful consideration of the contexts in which the methods and tools are to be applied.

4.1 "Off the Shelf" Product Development

Product developers can see the potential benefits in recording design rationale. Proceeding under time pressure, projects add new personnel, including programmers and members of support groups involved in documentation, training, quality assurance, performance analysis, and marketing. The education of new arrivals can be extremely time-consuming for existing team members; it can be very frustrating when new arrivals challenge design decisions that were made earlier for reasons lost from memory. Major productivity gains could plausibly result if a record of design rationale handled much of the education, freeing existing team members to continue working as developers. The design rationale could subsequently assist in the development of new versions or releases.

However, commercial products are generally failure-prone. According to Business Week, "estimates of new products' failure rate vary hugely, but that rate could be anywhere from 66% to almost 90%. In a 1991 survey, marketers expected 86% of their new products to fail, up from 80% in 1984." ("Will it sell?", 1992.) Reliable figures for commercial software projects are difficult to find, but in an informal poll of several experienced development managers from an array of computer companies, completion rate estimates ranging from 10% to 50%. Experienced developers reported having never seen a project through to completion and a well-known consultant recalled involvement in only one successful project.

Projects are terminated for different reasons. A companies might initiate competing projects with the intention of terminating one (see Kidder, 1982 for a

well known example). Companies change strategic direction or decide that a window of opportunity has closed. Schedule or budget overruns shake management's confidence in a development team. Early field testing reveals that a product will not recoup the cost of launching it. The list goes on. Note that the first three reasons provide clear incentives for developers to push ahead quickly: To beat internal competitors to milestones, to get something out and capture market share before a window closes, to meet project milestones and reassure management. These are upstream piranhas.

In this climate, it seems sensible to limit upstream activity to proofs of concept and to push toward completion. Resistance to expanding upstream activity may reflect the awareness of project managers of the precariousness of their enterprise.

The capture of design rationale in product environments is further hindered by the distributed nature of design decision-making. Managers, programmers, domestic and international marketers, human factors engineers, technical writers, industrial design engineers, mechanical engineers, training developers, performance analysts, and others are involved. Contributors to the design often work at different sites on different platforms. Conklin and Yakemovic, 1991 [Chapter 14 in this book]) encountered this problem when marketing and other groups would not adopt the development group's design rationale representation format. Not only wouldn't others use the system, the development group found it necessary to manually convert their design rationale into prose before meetings. This should be unsettling to those who envision design rationale as *contributing* to communication on a project!

A third obstacle is that the product developers who have to record their reasoning are unlikely to be the principal users of the rationale. Buyers of "off the shelf" products are largely responsible for adapting them. Vendor companies provide limited assistance, but it is delivered through customer support or field service groups—not by the initial development team. True, a company might derive an overall benefit by providing Customer Service with the development rationale, but Development and Customer Service budgets are distinct and such "altruism" is often absent (Grudin, 1991b). A chronic cause of failure in group work situations (Grudin, 1988; 1994) occurs when one group incurs additional work (in this case, the developers) and another group benefits (in this case, customer service).

Similarly, the initial development team often does not revise a successful product. New releases can be seen as less challenging, a form of maintenance. Product enhancements are often turned over to a subset of the team or to a different group as key designers are given new assignments.

Thus, the advocate of design rationale must ask a developer to undertake an effort that will slow down the project, be discarded unused in the likely event that the project is terminated, and in the best outcome will benefit someone else. And when a project is canceled, it will be natural to wonder: Might we have progressed more quickly and escaped the ax if we avoided that side effort?

Convincing examples of successes with design rationale are needed to counter these concerns. Until then a developer has little to go on.

Some assumptions underlie this bleak picture. Perhaps design capture could provide immediate benefits to the developer. Perhaps the effort required to explicitly structure design arguments could be minimized. Perhaps benefit could be obtained from a record of design rationale left by a terminated project. So far, immediate benefits have not been shown and existing systems require substantial effort to capture design rationale. In a unique case in which builders of a design rationale tool captured their own design rationale, described in Fischer et al. (1991 [Chapter 9 in this book]); the group ended up pessimistic about the approach. Chapters of this book reporting substantial trials (Shum, 1994 [Chapter 6]; Potts, 1994 [Chapter 10]) report difficulties in using existing methods to record rationale. The unlikelihood of benefiting from rationale captured in the course of a failed project—of using the rationale to learn from failure—is described in section 5.

In sum, much design rationale research is done in the product development context and is presented to audiences of product developers. The concept of design capture appeals to them when they focus on the positive possibilities. Rational approaches to problem-solving appeal to engineers. It is attractive to individuals who envision themselves as particularly effective users of such tools. Unfortunately, product development is also marked by tight time constraints, frequent progress monitoring, low project completion rates, and developer ignorance of post-sale use. These render product development a relatively unpromising context for exploiting design capture.

4.2 "In-house" or Internal Development

When an information systems group within a large organization, such as a bank, hospital, or automobile manufacturer, undertakes a development project, conditions are unlike those in product development. The system or application is usually large and focused on organizational support rather than individual use. Although management usually initiates a project with some specific functional goals, an internal development project often has a broader charter to address both functionality and interface issues than does a product development team formed after most functionality has been defined.

Are internal development project completion rates higher than in product development? Statistics are difficult to find, but there are disquieting indications. One survey reported that 75% of in-house projects are terminated prior to completion or produced software that was never used, the in-house equivalent of an unmarketed product (Gladden, 1982). In such a setting, a general policy of upstream investment to facilitate maintenance could be difficult to justify.

Positive factors exist for design capture in internal development. In-house projects are notorious for running over schedule, but their relative duration motivates the recording of design information as a hedge against losses of individual and corporate memory. As in other large projects, participants join a project as it progresses, requiring education about the rationale for past decisions.

Also, the same organization and perhaps the same individuals will be responsible for system maintenance, and a system or application is likely to be operated and maintained for a particularly long time. Large organizations often use systems for decades and are less motivated than product developers to make major changes that would require retraining users.

It may be difficult to introduce design capture tools and methods in this context. Tools available to in-house developers are usually older. Innovation is not attributed to such environments; innovation in the use of design capture is unlikely to be a widespread exception. Nevertheless, opportunities might present themselves to vigilant advocates of design capture.

4.3 Competitively-bid Contract Development.

Large contracted development projects have high completion rates, since both parties have strong incentives to avoid a cancellation. Delivered systems are often unusable without substantial further work (Martin, 1988), but corrective maintenance argues *for* supporting downstream activities. In addition, not only must the contractor educate new project members as development proceeds, but the delivered product is often maintained by people who have no access to the initial developers. Attributions of a high percentage of project costs to maintenance come from contract projects, where cost accounting is more routine. The high completion rate of contract development projects explains the inattention to project mortality considerations in the literature.

Also conducive to design capture in competitively-bid contract development is the detailed project documentation that already is required for the written specifications. The trend in government procurement is toward requiring a better record of design decisions and rationale: Concurrent engineering (CE) and computer-aided acquisition and logistic support (CALS) are new approaches that aim to integrate development and to allow the tracing of design features to specific requirement specifications. This perspective was shared by several participants in the AAAI'92 Workshop on Design Rationale Capture and Use.

However, there is a major disincentive to focus on upstream investment in large contract projects: The design is often carried out under one contract, the development of the resulting design is awarded in a second contract, and the particularly lucrative system maintenance can be carried out under a third. Each contract is let independently, potentially to a different organization. And even if the same company wins successive contracts, different people inevitably are assigned to complete them. This fragmentation of development effort can eliminate the incentive for those engaged in the upstream activities to allocate resources for supporting downstream activity. The contracting agency can try to mandate such support, but it is an open question whether useful compliance will result, especially given the experimental status of tools and approaches to capturing design information.

4.4 Customized Software Development

Smaller contract development efforts in specific market niches are particularly promising arenas for design capture, as well as being increasingly important centers of system development in general. As before, with a contract involved, albeit less formal and with less detailed specifications, these projects are relatively unlikely to be canceled. But in addition, those developing a systems for specific customers in particular markets are more likely to work with a customer after implementation, participating in various forms of maintenance (corrective, enhancement, etc.). (In contrast, close, long-term relationships with customers are often impossible in competitive procurement, where barriers are erected to prevent subsequent favoritism to one development company and where adherence to the letter of the contract can take precedence over the usability of the system.)

Another positive factor is that a consulting company that targets a specific application domain will seek similar projects that allow the reuse of software and domain expertise. A record that captures the initial rationale plus exceptions on subsequent projects could be very useful in a new project, as well as in maintaining existing systems.

A major hurdle to applying design capture techniques in this context is that many of these are smaller companies with fewer resources to invest in using and developing new or experimental tools. However, there are large consulting firms, and companies operating in specific markets are growing in size, number, and profitability.

Fischer et al. (1991 [Chapter 9 in this book]) explore the creation of design environments that support reuse in a manner conducive to customized software development, although they have thus far developed tools to support design activities outside of software (network administration, kitchen design, etc.).

5. WHAT CAN BE GLEANED FROM A TERMINATED PROJECT?

It has been argued that "the best prototype is a failed project" (Curtis, Krasner, & Iscoe, 1988). Project members learn from their experiences, whatever the outcome. Could design capture be exploited if employed in a prematurely terminated project? Key decisions affecting success could be reexamined and progress could be exploited. Design possibilities that missed a "window of opportunity" on a project could be more reliably or accessibly preserved for use in subsequent projects than occurs through the memory of individual participants.

This seems worth exploring in situations where a similar project is likely to follow a failure. But some hard questions have to be asked. Would people make the effort to record design rationale with this purpose in mind? Would such records be used? If created and used, would such records be more useful than project members' memories? Having experienced numerous project terminations, I am skeptical. The emotional aftermath of project terminations is not conducive to the use of project records. A dead project is about as inviting to linger over as the body of a victim of a highly contagious disease.

Projects are not terminated casually. A project can undergo major shifts in direction, schedule, budget, and even management without being terminated. Many factors may contribute to a termination decision. Precise reasons for a

cancellation may not be announced, but it is fair to assume that the product design was not brilliant. Any decision might have contributed to the failure in some way. A clear determination of responsibility may be difficult or impossible to reach; if reached, it may not be announced; if announced, it may not be accepted or believed. A pall descends over a terminated project. If a similar effort is initiated, the new management is keen to differentiate the new project, to put its own stamp on things. The desire to avoid overt associations with the decisions of the past can be superstitious (yet no less potent for that reason), but it can also be quite rational. Both symbolically and objectively, the new is dissociated from the tried and apparently untrue.

At the level of the individual contributors, disappointment and uncertainty follow a cancellation. Management works to move quickly past this state and to avoid extended recriminations. Public postmortems are rare. Less is learned from the experience as a result, but this cost is outweighed by the benefits of quickly confining skeletons to closets. The paucity of detailed analyses of failure testifies to this dynamic.

Individuals who were involved in a terminated project retain the ideas that particularly impressed them. Failed projects contribute when dispersed project members reintroduce certain points on their merits. Documentation from old projects is rarely explored systematically for ideas. Quite the contrary, in my experience the tangible traces of failed projects are discarded remarkably quickly.

This is true in research as well as development environments. At the research consortium MCC, one of the most frequent requests from shareholder companies was that researchers document not only their successes, but also their failures: paths tried and abandoned. The shareholders did not find it stressful to contemplate the examination of our failures, but we researchers did. We could not bring ourselves to make this effort. Focusing on the positive was far more appealing.

It could be argued that a design capture system provides the best of both worlds, documenting productive as well as abandoned approaches. However, where termination is frequent, a design rationale for an abandoned project will either be discarded quickly, as happens with other forms of documentation, or serve as a persistent reminder of failure. I cannot envision an ongoing practice of recording design rationale in termination-prone environments. Can the effortful documentation of failure be made palatable to those who must do the work to document their own failure?

At issue is the nature of what is possible in these emotionally charged situations, not the benefits that can plausibly be argued to exist. A reviewer of this paper asked “If a failed project bequeaths a rationale, then why couldn't the rationale live on?” It could, of course, but who would trust a rationale that supported a failure? And more importantly, who would help build a rationale if their prior experience suggested it would be a record of failure? Retrospective reconstructions of successful development projects are common; thoughtful

accounts of what was learned from failure are almost non-existent. Ignore human nature at your peril.

6. CUMULATIVE SCIENCE AND ENGINEERING TRADEOFFS

Many approaches to capturing and using design rationale have as an explicit goal the reduction of irrationality and inefficiency in design decision-making. The approaches are developed by researchers with scientific training and incorporate models of rational decision-making that resemble a scientific ideal. A full range of alternatives is explored, evidence is accumulated, a tentative decision is reached, and a record of activity is preserved for later reexamination. Whether or not science always proceeds this way, science is cumulative, progress is recorded and preserved, and thoroughness is valued. Engineering values are not in direct opposition to these, but they are not the same. Scientists are not designers. There is a clear danger that in developing tools to support a process, scientists will assume the process resembles that of science more than it does and fail to address the needs of developers.

Scientific exploration is often commended for its own sake. In engineering, exploration is not commended for its own sake. In a world of constant compromise, a point is reached sooner rather than later at which exploration is curtailed to save time or other resources. Those who have designed in areas untouched by standardization are aware that the smallest design decision, if examined closely and imaginatively, can open up a virtually infinite realm of possible solution paths. Setting out to chart a large unexplored territory can be good science; it is usually poor engineering. Scientific territory that is explored and reported augments a body of knowledge, but an open-ended engineering exploration diverts critical resources, misses windows of opportunity, and risks a termination that will leave no traces.

Scientists strive for certainty and replicability, so thoroughness is often commended for its own sake. Engineering tradeoffs often force decisions based on much lower probabilities, or even in the face of evidence.

Several of the empirical chapters in this book support this analysis. Shum (1994, [Chapter 6]) observed that “most decisions... may in fact not be of interest to other domain experts, because they are... not contentious given a certain level of expertise.” This observation is further reflected in Potts’s (1994 [Chapter 10]) distinction between a scientific “bird’s-eye view” looking down at design in its entirety and an engineering “turtle’s-eye view” rooted in the artifact at hand. Potts finds the latter “much more useful in the practical work of designing.” Lewis, Rieman and Bell (1991 [Chapter 5]) take this a step further by arguing for design rationale that is entirely problem-based.

The careful recording of the rationale underlying design decisions can be useful to engineers. But the value is a function of the tradeoffs and compromises that constitute engineering practice, which often seem to be ignored by innovators who hope to influence design. If the tools and methods designed by researchers with a scientific background reflect too strongly the values and ideals of science, those tools and methods will be less useful to engineers.

7. CONCLUSION

The creation of a record of design decision-making, a design rationale, has a strong intuitive appeal. The current level of interest in exploring it is a positive development. For such efforts to enjoy the best prospects for evolving into useful tools and methods, it is important that they be designed with particular use situations in mind. Different development contexts require different approaches.

Several obstacles to the capture and use of design rationale in software development have been identified. It can slow down a project that is under pressure to reach a milestone. It can become a record of failure. It can require a joint effort by people separated by time, geography, platform, and even employer. The effort can be lost in the chaos of a terminated project. It can require commitment to a new, experimental approach. To overcome these obstacles, design rationale systems must provide considerable collective benefit. Those who have to do the work of recording design information should perceive themselves as benefiting from the use of the record. On the positive side, some software development contexts provide relatively promising conditions for such efforts, notably customized software development.

Gruber and Russell (1994 [Chapter 11 in this book]) describe the need to look carefully at prospective users of design rationale to see what they need. Adopting this approach would quickly reveal some of the impediments described in this chapter. However, generalizing across cases must be done sparingly; such an analysis should be considered for each project.

This chapter provides suggestions, not definitive answers. These are research issues, to be explored in parallel with the development of tools and techniques for the capture and organization of design rationale.

Acknowledgments. General discussions with Gerhard Fischer and the HCC group at the University of Colorado and specific discussions about customized software development with Frank Shipman were helpful. Jack Carroll, Tom Moran, and an anonymous reviewer contributed very helpful comments on an earlier draft.

Support. The preparation and writing of this paper were supported by the National Science Foundation under grant No. IRI-9015441.

REFERENCES

Boehm, B. (1981). *Software engineering economics.* Englewood Cliffs, NJ: Prentice-Hall.

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

Carroll, J. M., & Rosson, M. B. (1991). Deliberated evolution: Stalking the view matcher in design space. *Human-Computer Interaction, 6,* 281-318. Also in T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. [Chapter 4 in this book.]

Conklin, E. J., & Yakemovic, KC B. (1991). A process-oriented approach to design rationale. *Human-Computer Interaction, 6*, 357-391. Also in T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. [Chapter 14 in this book.]

Conway, M. E. (1968, April). How do committees invent? *Datamation,* 28-31.

CSTB (1990). Scaling up: A research agenda for software engineering. *Communications of the ACM, 33*, 281-293.

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

Fischer, G., Lemke, A. C., McCall, R., & Morch, A. I. (1991). Making argumentation serve design. *Human-Computer Interaction, 6*, 393-419. Also in T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. [Chapter 9 in this book.]

Fischer, G., Grudin, J., Lemke, A. C., McCall, R., Ostwald, J., Reeves, B., & Shipman, F. (1992). Supporting indirect collaborative design with integrated knowledge-based design environments. *Human-Computer Interaction, 7*, 281-314.

Gladden, G. (1982). Stop the life-cycle, I want to get off. *Software Engineering Notes, 7,* 2, 35-39.

Gruber, T. R., & Russell, D. M. (1991). Generative design rationale: Beyond the record and replay paradigm. In T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates. [Chapter 11 in this book.]

Grudin, J. (1988). Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces. *Proc. CSCW'88*; 85-93. New York: ACM. Reprinted in D. Marca & G. Bock (Eds.), *Groupware: Software for computer-supported cooperative work* (pp. 552-560). Los Alamitos, CA: IEEE Press, 1992.

Grudin, J. (1991a). Interactive systems: Bridging the gaps between developers and users. *IEEE Computer, 24*, 4, 59-69.

Grudin, J. (1991b). Systematic sources of suboptimal interface design in large product development organizations. *Human-Computer Interaction, 6*, 2, 147-196.

Grudin, J. (1994). Groupware and social dynamics: Eight challenges for developers. *Communications of the ACM, 37*, 1.

Jarczyk, A. P. J., Loffler, P., & Shipman, F. M. (1992). Design rationale for software engineering: A survey. *Proceedings of the 25th Hawaii International Conference on Systems Sciences,* 577-586. Los Alamitos, CA: IEEE Computer Society Press.

Kidder, T. (1982). *Soul of a new machine.* New York: Avon.

Lewis, C., Rieman, J., & Bell, B. (1991). Problem-centered design for expressiveness and facility in a graphical programming system. *Human-Computer Interaction, 6,* 319-355. Also in T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. [Chapter 5 in this book.]

Martin, C. F. (1988). *User-centered requirements analysis.* Englewood Cliffs, NJ: Prentice Hall.

MacLean, A., Young, R. M., Bellotti, V. M. E., & Moran, T. P. (1991). Questions, options, and criteria: Elements of design space analysis. *Human-Computer Interaction, 6,* 201-250. Also in T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. [Chapter 3 in this book.]

Potts, C. (1994). Supporting software design: integrating design methods and design rationale. In T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates. [Chapter 10 in this book.]

Sharrock, W., & Anderson, R. (1994). Organizational innovation and the articulation of the design space. In T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates. [Chapter 15 in this book.]

Shum, S. (1994). Analyzing the usability of a design rationale notation. In T. P. Moran & J. M. Carroll (Eds.), *Design rationale: Concepts, techniques, and use* (pp. xxx-xxx). Hillsdale, NJ: Lawrence Erlbaum Associates. [Chapter 6 in this book.]

"Will it sell? Hard to tell." (1992, August 17). *Business Week,* p. 72 B-E.