SYSTEMATIC SOURCES OF SUBOPTIMAL INTERFACE DESIGN IN LARGE PRODUCT DEVELOPMENT ORGANIZATIONS

Jonathan Grudin
Aarhus University

ABSTRACT

Many poor interface features are the result of carelessness, ignorance or neglect in the development process. For these features, methods such as user involvement in iterative design with prototyping, the use of checklists and guidelines, and even formal evaluation can be of great help. However, there are strong forces present in development environments that block the use of such methods and distort interface designs in a systematic way. Because these forces serve legitimate goals, such as making a design simpler, more easily communicated, or more marketable, they are more difficult to counter; because developers are skilled at working toward those goals, the tangential effects on the interface usually pass unnoticed. This descriptive, empirical article describes these forces in the context of large organizations developing commercial off-the-shelf software products. Most points are supported by examples and by a logical argument. Not all of the phenomena may appear in a given development organization, but the overall picture of a complex environment in which interface development requires unwavering attention is quite general.

Author's address: Jonathan Grudin, Department of Computer Science,
Aarhus University, Ny Munkegade 116 Building 540, DK-8000 Aarhus C, Denmark

CONTENTS

1. INTRODUCTION

The need for developers of interactive systems to understand the requirements of the eventual users and the work that the systems are to support is well known. Developers obtain this understanding in different ways. They often rely on intuition. They also learn about users indirectly, by reading or by being told about users' needs. They may have direct but limited interaction with users through informal interviews, laboratory tests of prototypes, and other means. The most embracing approach, sometimes called *participatory* or *collaborative* design, enlists prospective users as full members of the development group.

There is a strong consensus that intuition and indirect approaches to understanding users and their work are usually insufficient. The design principles formulated by Gould and his colleagues at IBM (summarized in Gould, 1988) are: (a) focus early and continuously on users, (b) integrate consideration of all aspects of usability, (c) test versions with users early and continuously, and (d) iterate the design. Despite being widely cited, these principles are not often followed. Many of the useful and usable systems that exist are the result of an undesirably long evolutionary process. Gould wrote that the principles "are hard to carry out, mainly for organizational and motivational reasons" (p. 776), which are not specified. This article describes these reasons.

My description focuses on one systems development context: large product development organizations. Product development organizations and the development projects within them are described in the next section. Most of the companies described were formed before product usability attained its present visibility. As a result, they did not address the particular needs of developing interactive systems when defining their basic organizational structures and development processes. Section 3 of this article outlines the consequences: common difficulties in achieving and benefiting from user involvement in development. In Section 4 goals that are present in such organizations and that can conflict with good interface design are described. These goals often seem unrelated to interface considerations, but in the absence of knowledge of users and their work, they can be an unrecognized, subtle source of bad design decisions. In Section 5, I describe approaches to addressing the problems. Eventually, organizational change may be required. In the meantime, those working within

such organizations must be aware of the problems and seek constructive paths around them.

Because goals that conflict with interface optimization are always present, interface development requires constant vigilance. Poor design features that result from carelessness or ignorance are dispelled by applying care or knowledge, but poor features resulting from competing pressures often recur in the same or in different locations.

This article draws on the growing literatures in human-computer interaction, much of which originates in product development companies. It also relies on surveys and interviews of over 200 interface designers in several product development companies (Grudin & Poltrock, 1989; Poltrock, 1989b), experiences in product development, and thousands of conversations with fellow developers over the years. Of course, organizations vary considerably. Reliable, industry-wide data are difficult to find. The obstacles described here are encountered, but not universally. The hope is that the forewarned reader will be better able to anticipate, recognize, and respond to these and similar challenges, if and when they arise.

2. THE CONTEXT: PRODUCT DEVELOPMENT PROJECTS

This section contains an outline of the development context discussed throughout the article. First, large product development organizations are defined. Then, development projects within these organizations are analyzed, revealing why the human-computer interface or “user interface” is the principal focus of utility and usability for a product development team. Finally, the range of possible user involvement, from full collaboration to occasional or indirect consultation, is discussed.

2.1. Product Development Organizations

This article focuses on large organizations that develop and market “off the shelf” or “shrinkwrap” software applications and systems. This excludes in-house development projects and projects undertaken to fulfill contracts. Internal and contract development have different advantages and disadvantages for user involvement in developing interactive systems (Grudin, 1991a). Of course, companies often straddle categories: a product development company that bids on government contracts, a company that markets a system built initially for internal

use or under contract, and so forth. In addition, small product development companies may not experience the problems described here, and companies of moderate size are likely to experience some and not others.

Although product development is only a fraction of interactive systems development, it is the locus of much advanced interface work. Large product development companies are visibly concerned with usability and "look and feel." They hire and train many user interface specialists, recruiting heavily from research universities. Such specialists dominate the conferences and journals in the field of human-computer interaction, especially in the United States, as reflected in the strong presence of product developers from IBM, Digital, Hewlett-Packard, Xerox, Apple and other large companies at Computer and Human Interaction (CHI) Conferences.

Most of these companies matured in the 1960s and 1970s. Their principal source of revenue was selling or leasing hardware; software functionality was secondary, and the human-computer interface received little attention. Most processing was batch, not interactive, and the immediate users were computer professionals. Since then, software has come to rival hardware in importance and interactive software is widespread; many of the successful new product development companies of the 1980s primarily sell interactive software. Until the success of the Macintosh in the late 1980s, the focus was entirely on functionality and price; now, the interface is increasingly important.[1]

Although attitudes are changing, the business operations and development practices of most of today's large product development companies were formed when hardware and software functionality were the only considerations. It is not surprising that their basic organizational structures and processes do not facilitate interface development. In fact, we will see that the design and development of good interfaces are often systematically *obstructed*—not intentionally, but obstructed nonetheless.

---

[1] Apple differs from most large product development companies in several ways. In particular, it matured more recently, in the era of interactive systems, and perhaps for that reason does not share some of the organizational structures and development practices that are described here.

2.2. Development Projects: Defining Function before Form

The timeline of an ongoing development project includes a start date and a projected completion date. Reality is not always very crisp; one event flows into another, and decisions are gradually recognized to have emerged along the way. However, the project start date does have meaning: A team is formed, assignments announced, and budgets allocated. The start date may even be dramatized, as when the first working name of a project is a number based on the month and day of its initiation. Easily overlooked is the activity that preceded the start date.

Product definition precedes the formation of the development team. During this phase the high-level "functionality" is identified. In the second phase, the team is assembled, and it designs and develops the necessary low-level operations, the visual appearance, the documentation, and other aspects of the human-computer interaction. Loosely speaking, the functionality is defined in the first phase and the interface in the second. Although in principle it is notoriously difficult to draw a line between software functionality and its interface, a de facto distinction is made for each project. The high-level functionality is defined by a different group of people, based on a wide range of factors, before the development team is formed. The development team is handed the product idea and is responsible for the remaining design.[2] The precise division of design responsibility varies from project to project. In fact, as the interface grows in importance, key design decisions move from the development phase to the product definition phase. A product is required to have a graphical interface, adhere to a corporate "look and feel," run on Windows 3.0, and so on—constraining the design space of the development team.

When can user involvement enter this two-phase process? First, consider the possibility of user participation in product definition. This phase is generally

---

[2] In contrast, those working on in-house development projects avoid the functionality/interface split to the degree they consider the system as a whole. It makes sense to do so; one can have a usable system that is not very useful or a useful system that is not very usable, but the most desirable outcome suggests working on both together. (Of course, this requires resisting a premature definition of the proposed system based on management and perhaps worker preconceptions.) Thus, researchers and developers working on in-house systems, including the British and Scandinavian participatory design experiments, focus on the work that the computer is to support and find the "user interface focus" of the CHI or human-computer interaction field to be limited. The latter focus is natural for product developers, whose involvement generally follows product definition.

carried out by a management group. Companies vary in the degree to which they are driven by engineering or marketing initiatives, but marketing typically influences product definition. "User needs" are considered in developing a market justification or business plan. Open-ended needs-finding—involving users without any preconceptions—is not unheard of, but it is rare; ideas for products or enhancements are usually plentiful. Other factors come into play: Does the potential product fit coherently into the existing product line? Will it undermine sales of other products? Are major competitors using a feature to make inroads into the company's existing customer base? Does the marketing and sales force have the expertise and motivation to sell it? Will sales be enough to pay for the necessary advertising campaign? Does it test a potentially desirable new application domain for the company? Is it a new release needed primarily to reassure customers that the product line is not being abandoned? Strategic decisions such as these are crucial to the company and are often closely guarded for competitive reasons. Although major customers are consulted, the possible role of user involvement at this point is very limited.

As the product definition phase proceeds, issues that would benefit from such involvement may be addressed. The market analysis may include surveys and focus groups. Often, mediators—consultants, the trade press, internal marketing staff, users' "representatives" (e.g., systems analysts who also represent the interests of others, such as the users' management, which may not be identical) are relied on for information about potential users. But "end users" from customer organizations are unlikely to be major participants in this complex product definition process. After all, rarely are individual developers from the development organization fully involved.

The rest of this article focuses on the period after the functional specification is defined and the baton is passed to the development team. Once the project team is assembled, the product definition group largely recedes from view, moving on to other concerns while monitoring progress through documentation and management reviews. This strongly phased process resembles the requirements-driven model that originated in contract development and is widely applied to internal development (Grudin, 1991a). The belief that a written requirements specification is enough to communicate a product idea had more validity for noninteractive systems; applying it here may result in the problem found in contract and internal development: A product is delivered that is not quite what

those defining it had in mind or what its recipients find usable. As product usability draws more attention, more rapid-prototyping efforts may occur within marketing groups prior to product definition, permitting more opportunity for collaboration among marketers and developers—and perhaps users.

The part of the design affecting the eventual users that remains to be defined by the development team corresponds approximately to the human-computer interface, liberally defined. This includes documentation, training, hot-line support, and other elements that directly affect the users' experiences with the product. It also includes low-level functions; for example, whether a one-step "move" or a two-step "cut" and "paste" is provided. The development team may have license to add features that can be justified. Conversely, as just noted, some aspects of the interface may have been constrained in the product definition phase.

In summary, the nature of product development works to separate the definition of high-level functionality from subsequent development. For this reason, user involvement in product development projects is almost exclusively in furtherance of the design and evaluation of the interface.

2.3. Degrees Of User Involvement

What is the optimal degree of user participation in development? If you are developing a compiler, users' involvement will be minimal. If you are copying features from an existing product in a mature application area, limited contact with potential users can be adequate. If you are developing an interactive system in a new domain, full collaboration with users can be essential.

Here we find an important contrast to the in-house development situation. In-house projects involve the development of a new system for a specific group of users. Because the system must be accepted by the target group, a compelling argument can be made to learn as much as possible about the future users' shared or even individual backgrounds, work practices, and preferences. And the socio-technical and participatory design approaches arose in this context. The product situation is different; the specific users are not known in advance, one is in a sense targeting a "greatest common denominator." Earlier versions or competitors' products often serve as guides; as already noted, many of the major decisions were made in the product definition phase.

Product developers also have a particularly wide range of *indirect* approaches to acquiring some understanding of computer users. They learn about users in courses, conferences, and trade shows. They rely on marketing and sales people, customer service and training staff, consultants, information systems specialists, users groups, standards organizations, trade magazines, and journals. The adequacy of these and other indirect approaches depends on the circumstances, but many of these mediators exist to serve this communication role and have an incentive to make themselves useful.

Although there is no universal answer or easy algorithm for calculating the optimal degree of direct user involvement, current practice seems to produce too little. Many products are unusable or unnecessarily difficult, and strong forces are pushing developers to obtain more detailed information about future users of their products. Four of these forces are:

1. The spread of computer use into environments that are increasingly unlike development environments. In the past, most people who interacted directly with computers were engineers, programmers, or trained operators, so developers' intuitions about user environments were good. As computer users become less technical and more diverse, developers acquire a corresponding need to obtain more information about them and their work environments.

2. The rising expectations of computer buyers and users. Even as the growing capabilities of applications and systems present greater challenges in instruction and information presentation, the willingness to adjust to the system and the tolerance for poor interfaces are declining. As product prices decline, heavy investment in training is less palatable. Again, the burden shifts to the developers to create responsive systems.

3. Maturing application areas. In a new market, functionality is likely to govern purchasing decisions. But when several products offer comparable functionality, ever more detailed knowledge about users and the contexts of use is needed to fine-tune products and provide an edge.

4. Emerging applications that support groups ("groupware") require more knowledge of users and their environments than did single-user products. Groupware must support a wider range and greater percentage of users in a given setting, bringing product developers closer to the situation that in-house

developers have faced. The limitations of any one person's intuitions for group behavior and a greater need to consider issues surrounding adoption in the work environment require developers to gather more information than before (Grudin, in press).

These escalating demands, as well as frequent product failure and user dissatisfaction, suggest that intuition and indirect methods of learning about users, which are not doing well now, are likely to be increasingly inadequate. Next, we consider direct user involvement.

Direct contact takes many forms. Actual or potential users may be observed, surveyed, interviewed, tested, studied, and experimented upon. Remarkable ingenuity has been displayed in finding ways to poke and prod users. Can approaches that fall short of full collaboration work? In in-house development projects, charged with addressing the specific needs of a well-defined set of users, full collaboration may be necessary. In product development, lesser measures can answer *some* questions. Product developers, aiming for a broad market, are less concerned with differences than with commonality. They have focused on characteristics shared by most users: motor skills (keyboard layout, mouse control), perceptual processes (character legibility, color contrasts), and cognitive processes (recognizing menu item names, scanning displays for information). Limited forms of user involvement such as formal experiments resolve some of these questions. However, to deal with issues closer to the task or application domain, these laboratory methods are less adequate. In fact, even low-level design issues such as function key placement and default menu item assignment can hinge on specific aspects of users' tasks (Grudin, 1989).

A full collaboration with future users may be preferable to attempts to "extract information" from users late in development. Consider the situation turned around: Imagine an "end-user programming" setting in which the developers play a secondary role. The system is largely "tailorable" and the "project team" consists primarily of users, with programmers in a support role. In this example, we would argue that the team of users should attempt to keep the programmers involved from the start and not hope to rely on extracting information from them when some coding is necessary in the late stages of redesigning the system!

Product developers can be aware of the virtues of heavy user involvement in development; for example, they may work with domain experts hired from user organizations. Gould and Lewis (1983) of IBM made an early, forceful argument for participatory design, eschewing reliance on mediators and more limited empirical approaches:

> We recommend that typical users (e.g., bank tellers) be used, as opposed to a "group of expert" supervisors, industrial engineers, programmers. We recommend that these potential users become part of the design team from the very outset when their perspectives can have the most influence, rather than using them post hoc to "review," "sign off on," "agree" to the design before it is coded. (p. 51)

This advice has been prominently republished (e.g., Gould, 1988; Gould, Boies, Levy, Richards, & Schoonard, 1987; Gould & Lewis, 1985) and is widely cited, yet it is rarely followed. Gould and Lewis (1985, p. 304) allude to "obstacles and traditions" that stand in the way. The next sections explore those obstacles and traditions.

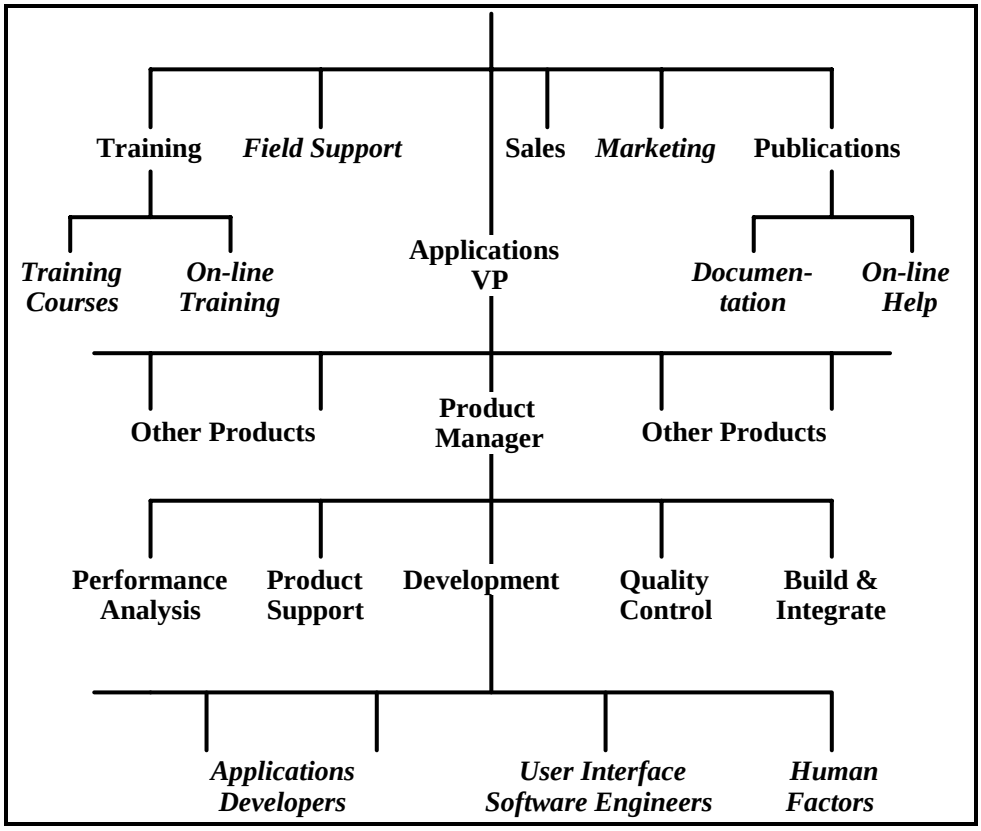
3. OBSTACLES TO USER INVOLVEMENT

This section covers challenges to user involvement that arise in several ways. Some are due to the inherent nature of product development: The actual users are not truly identified until development is complete and the product is marketed, potential users work for different organizations, and any one set of users may be too limiting. Additional obstacles can be traced to the division of labor within development organizations that were often established to develop and market non-interactive systems. Typical allocations of responsibility serve useful purposes, but they distribute aspects of the interface across organizational boundaries and separate software developers from the world outside (see Figure 1).[3] In particular, contact with customers and users is the province of groups or divisions outside of development: sales, marketing, training, field support, and upper management. The people assigned these tasks are not primarily concerned with the interface, their relevant knowledge is not systematically organized, and they are often

[3] Many small and large variants of this organizational structure are found. For example, functions such as quality control and performance analysis may be handled centrally.

located far from the developers. They have a limited sense of what information would be useful or to whom they should forward it. Finally, after discussing these structural impediments, I examine difficulties that can be traced to standard software development procedures and techniques.

Figure 1. **A typical organization chart (partial) showing the separation of user-related functions.**



3.1. Challenges in Motivating the Developers

For user involvement to be effective, most or all members of the development team must be committed to the approach. One person can work with users and try to introduce the results into the process, but iterative development requires broad involvement, prototyping and testing often require software support, and the results must be valued. Management must be willing to invest the resources, and the help of others may be needed to smooth contacts with users.

Although most developers would agree to user involvement in principle, they may not follow through for several reasons. Some engineers lack empathy or

sympathy for inexperienced or nontechnical computer users. When developers and users meet, they sometimes find that different values, work styles, and even languages get in the way of communicating. Developers tend to be young, rationalistic, and idealistic, products of relatively homogeneous academic environments. They often have little experience or understanding of the very different work situations and attitudes of many system users. The best of intentions can succumb to these factors, especially in the face of the slowness and imprecision that often accompany user involvement.

3.2. Challenges in Identifying Appropriate Users

Developers may have a market in mind, but the actual users of a product are not known until the product is bought. This is particularly true for new products, but even new releases or extensions of existing products are often intended to expand market share in one or another direction, and they may not be adopted by all existing users. The fates of many products, both positive and negative, are reminders of the inherent uncertainty in product development. The IBM PC had a wider than expected appeal, for example, and we can be confident that the developers of countless failed products anticipated users who never materialized.

Further obstacles to identifying potential users stem from the nature of developing products intended to appeal to a broad range of people. The effort is focused on casting as wide a net as possible; to shift to narrowing perspective in order to identify specific or characteristic users is difficult. Choosing one set of users risks ignoring other individuals or groups. The seriousness of the problem of defining characteristic users can be seen by considering the experience of Scandinavian researchers in the more favorable in-house development context. These projects began with relatively constrained user populations within one industry or even one organization. Even so, selecting representative users was found to be a major challenge (e.g., Ehn, 1989, pp. 327-358). Such problems are greater for developers of generic products that are widely distributed by the development company or by independent software vendors.

Obstacles also arise from the division of labor. User interface specialists rarely see “the big picture.” They may work with a development team assigned to a single application or even to part of an application. Not even the project manager has a perspective encompassing the application mix that customers are expected to use, the practices and preferences of the installed customer base, and

strategic information about the intended market for a product. Although this broad perspective may be found in marketing or sales divisions, these divisions are often geographically and organizationally distant from the development groups. The projected market—the identity of the future users—may be closely guarded by upper management due to its competitive importance.

In large companies, marketing and sales representatives become species of users of products emerging from development. They also consider themselves to be internal advocates for the customers. Because the customers are often information specialists or managers, rather than "end-users," the chain of intermediaries lengthens. Low levels of contact, miscommunication, and a lack of mutual respect between marketing and development can further reduce the value of this indirect link between developers and users (e.g. Grudin & Poltrock, 1989; "Mitch Kapor," 1987; Poltrock, 1989b).

Another complication in identifying appropriate users is that a system is often modified substantially after the development company ships it but before the users see it. This is done by software groups within customer organizations and by value-added resellers who tailor products for specific markets, for example. These developers are in a real sense users of the product—perhaps among the most important potential users. It may be more appropriate for *them* to involve the actual end-users. In any case, the initial development team must discover which aspects of their design are likely to be "passed along" to users. Third-party intermediaries take on some of the responsibility for meeting user needs, but their role complicates the selection of representative end-users.

3.3. Challenges in Obtaining Access to Users

Once candidates have been identified, the next challenge is to make contact with them. Obstacles can arise within the users' organization, within the development organization, or in the gap between them.

In the User Organization. Contacts with customers are often with managers or information system specialists, rather than with the computer users themselves. It may not be easy for developers to get past them: Their job is precisely to represent the users. In addition, the employers of prospective users may see little benefit in freeing them to work with an outside development group.

In the development organization. Within the product development company, *protecting (or isolating) developers from customers is traditionally a high priority.* The company cannot afford to let well-intentioned developers spend all of their time customizing products for individual users; priority is given to developing generic improvements to benefit scores or hundreds of users. Savvy customers are well aware of the value of having the phone number of a genial developer. Barriers erected to keep users from contacting developers also prevent developers from easily connecting with users: The relationships and channels are not there.

The development company's sales representatives may be reluctant to let developers meet with customers. A developer, coming from a different culture, might offend or alarm the customers or create dissatisfaction with currently available products by describing developments in progress. Similarly, the marketing group may consider itself the proper conduit into the development organization for information about customer needs and fear the results of random contacts between developers and users. In one company, developers, including human factors engineers, were prevented from attending the annual users' group meeting. Marketing viewed this as a show staged strictly for the customers.

Physical and linguistic barriers. User organizations may not be nearby—a formidable barrier to sustained user involvement. As international markets become increasingly central for many large product development companies, access to an important segment of the user population is cut off by linguistic, cultural, and physical barriers.

Sometimes a potential user is *within* the development company. This is convenient and can be used to good advantage, but it is a dangerous special case to rely on. The company is not in business to build products for itself, and user environments are less likely to resemble development environments than in the past. Of course, the riskiest user of all is the developer-user. The developer-user is generally aware of implementation details and is often an infrequent user who has detailed knowledge of the product, a very atypical combination of user characteristics.

3.4. Challenges in Motivating Potential Users

In an in-house development project, the product developers share the same management (at some level) with the potential users. Because this is not true in

the case of product developers and external users, it is more difficult for the users to obtain time away from their jobs. In addition, the potential users can be less motivated, knowing that they may not end up being actual users of the final product. The problems of sustaining user involvement have been recognized as substantial even in internal development projects. Scandinavian collaborative design projects have emphasized techniques for maintaining a high level of user interest (Greenbaum & Kyng, 1991). Of course, for contacts of limited duration, many computer users and their employers are pleased to be consulted.

Potential users can be less motivated if they do not see how the planned product will benefit themselves if they do get it. This is particularly problematic for large systems and for groupware applications, most of which require widespread use but selectively benefit managers (Grudin, 1988; 1990). The situation is even worse if the potential users feel that their jobs might be threatened by a product that promises increased efficiency.

3.5. Challenges in Benefiting From User Contact

As described in Section 2.3, contact between developers and users takes many forms. To benefit from verbal interaction, they must learn one another's ways of talking about technology and its use or develop a new language. The importance of this difficult task is also emphasized in recent Scandinavian projects (Greenbaum & Kyng, 1991). The use of prototypes can reduce the dependence on verbal communication, although the prototyping tool can introduce design constraints or fail to reflect future implementation constraints.

Contact with users inevitably provides developers with only a partial understanding of the use situation. In particular, although developers can obtain insight into the nature of users' first encounters with a novel system, application, or feature, patterns of highly experienced use can be very important and are more difficult to explore prior to product release. Even in projects with careful user testing of design features, developers override test results in response to the intuited needs of heavy users (e.g., Bewley, Roberts, Schroit, & Verplank, 1983). Because developers are likely to assume that heavy users will resemble themselves, this is a serious dilution of user involvement.

For user contact to be translated into user involvement, it must have an impact on product design. When the identity of future users is uncertain and a wide range of conceivable candidates exists, a team may find it difficult to assess

their experiences with a small number of possible users. The Scylla of overgeneralizing from a limited number of contacts is accompanied by the Charybdis of bogging down when users disagree. Again, evaluation is particularly challenging for systems that support groups (Grudin, 1991b). Finally, if user involvement does result in design recommendations, only the first step has been taken. Design ideas must be steered through a software development process that is typically fraught with obstacles to interface optimization. User involvement can increase the odds of successfully navigating this course, but the journey is rarely easy, for reasons described in Section 4.

3.6. Challenges in Obtaining Feedback From Existing Users

Feedback from users may be collected informally or through bug reports and design change requests. The latter generally focus on what is of primary importance in the marketplace (e.g., hardware reliability and high-level software functionality) and not on interface features. The little information that *is* collected rarely gets back to developers. Customer support groups such as training and field service shield developers from external contacts by maintaining products and working with customers on specific problems. The original product developers move on to new releases or product replacements, are reassigned to altogether different projects, or leave the company for greener pastures.

The extent of feedback may vary with the pattern of marketing and product use. A company such as Apple, with a heavy proportion of discretionary purchases initiated by users rather than by management or information systems specialists, accrues benefits from having a particularly vocal user population. In general, though, a lack of user feedback may be the greatest hindrance to good product interface design and is among the least recognized defects of standard software development processes. System developers cannot spend all of their time fielding requests from customers, but their overall lack of experience with feedback is an obstacle to improving specific products and to building an awareness of the potential value of user participation in design. Developers rarely become aware of the users' pain.[4]

---

[4] Because exposure to feedback is so rare, its effect on developers can be dramatic when it does occur. Developers are often motivated by seeing videotapes of users struggling with their product, after a brief period of blaming the difficulty on the users. ("I have a problem with this," one development manager interrupted a screening to say. "My problem is, I'm watching a moron." But

This point deserves emphasis. Engineers are engaged in a continuous process of compromise, trading off among desirable alternatives. Interface improvements will be given more weight if engineers are aware of the far-reaching, lasting consequences of accepting an inferior design. Consider some typical tradeoffs: "This implementation will save 10K bytes but be a little less modular." "This design will run a little faster but take 1 month longer to complete." "This hardware configuration provides two more slots but adds $500 to the sales price." Each tradeoff requires a decision. Once the decision is made, the price in development time, memory size, or chip expense is paid and the matter is left behind. In this environment, the interface is just one consideration among many. "This interface would be better, but take 2 months longer to design." The decision may adversely affect thousands of users daily for the life of the product, but without feedback, a developer remains unaware of this special characteristic of the interface. Once it is built and shipped, they are on to the next job, and other people (including users) must deal with any problems.

3.7. The Difficulty of Identifying Design Teams

With whom are users to collaborate? Despite recommendations that one group have responsibility for all aspects of usability (Gould, 1988), the "user interface," broadly defined, is not the province of one recognizable team in a large product development company. The hardware is designed by one group, the software by another, the documentation by a third, and the training by a fourth. Representatives from other groups have periodic involvement—reviewing design specifications, for example. A product manager with little direct authority over developers may coordinate scheduling. Individuals from several different marketing groups, such as competitive or strategic analysis[5] and international marketing, can contribute. Members of technical support groups, such as human factors or performance analysis, often participate, although not necessarily throughout the project. Several levels of management monitor the process and comment at different stages. Finally, turnover in project personnel can be a further obstacle to sustained user involvement.

---

after being told that the person had worked for the company for 5 years and that a majority of testers had the same difficulty, the manager was soon enthusiastically proposing interface changes.)

[5] Competitive analysis may seem to be a logical ally of a development organization. However, in practice its concern is often the effective marketing of existing products against competition, rather than the planning of future products.

In concert, these people contribute to defining a computer user's eventual experience with the system or application, yet communication among them is often surprisingly sparse. In Section 4, the divisions of responsibility in these organizations are discussed. Established to reduce unnecessary communication in a time of less demanding user requirements, the divisions can now create problems.

Poltrock (1989c) described a case in which lack of communication clearly affected usability. A new release of a popular product introduced an improved method for accessing certain functions. Although the developers planned to phase out the original, less efficient interface, they retained it as an alternative in this release to provide "backward compatibility" for current users. But the training development group was not informed of the overall plan and developed training materials that taught only the old, inferior method.

3.8. The Late Involvement of Interface Professionals in Projects

In a survey of interface designers from different disciplines within several large product development organizations, Grudin and Poltrock (1989) found late involvement in the software development process to be a widespread complaint among members of support groups. For example, 57% of the human factors engineers and 28% of the technical writers reported typically being involved in projects before implementation started, whereas 100% of the former and 87% of the latter said they would prefer to be involved that early. The human factors engineers estimated that for 39% of their projects involvement started too late; the technical writers estimated that 52% of the time they were involved too late.

In Section 4, historical and organizational reasons for late consideration of the interface are described. The project members most likely to advocate user involvement are those just mentioned. If management is unaware of the need for *their* early and continual involvement, how much support will their calls for early and continual *user* involvement receive?

3.9. Not Enough Time

The 200 interface designers responding to the aforementioned survey estimated that "insufficient development time" caused "substantial impairment" to over one third of all the interfaces developed—the greatest percentage of impairment attributed to any one factor. One cause of this damaging haste, of

course, is the competitive necessity or advantage in getting a product to market quickly. A second cause is the late involvement in a development project of the nonprogrammer interface professionals, as just described. These are clear barriers to extensive user involvement. And as a final "Catch 22," even late user involvement is blocked. Once the underlying software code is frozen, a fully functioning system is available that users could try—but at that very moment, documentation moves into the critical path toward product release. Because it is the software interface that is being documented, the interface is also frozen—before a user can try it out!

Poltrock (1989a) described in more detail the unique problems that high visibility and dependencies create for the interface development process. One developer summed it up:

> I think one of the biggest problems with user interface design is that if you do start iterating, it's obvious to people that you're iterating. Then people say, "How is this going to end up." They start to get worried as to whether you're actually going to deliver anything, and they get worried about the amount of work it's creating for them. And people like [those doing] documentation are screwed up by iterations. They can't write the books. Whereas software, you can iterate like mad underneath, and nobody will know the difference. (pp. 192-193)

The fear that user involvement will lead to unacceptable delay is a major impediment in the time-pressured product development environment, despite arguments that this does not happen (e.g., Gould et al., 1987). Even if some additional time is required, a product's likelihood of succeeding may be much greater. As recounted by Conway (1968, p. 29), "there's never enough time to do something right, but there's always enough time to do it over."

4. GOALS THAT SHAPE INTERFACES WHEN USERS' VOICES ARE NOT HEARD

Everyone in an organization is working toward many goals at any given time. These include personal goals, team or group objectives, and the purposes of the organization as a whole. Goals also enter the work environment that originate in industry-wide developments, in professional group activities, and in concerns of the local community and broader society. There are far more goals in an

organization than there are people—for a large development company, this means many goals indeed!

These goals do not all conflict, of course. Companies strive to see that employees share the corporate goals. Teams are established in part to foster common goals of a more focused nature. Ideally, people are working in concert or toward goals that have only incidental effect on the work of others. But inevitably, goals conflict. No one gets everything they would like. Adjustments happen. Discovering conflicts, arguing, compromising, horse trading, adjusting, re-prioritizing, and accepting is part of an ongoing process. Much of it goes on unnoticed—we have highly developed skills that allow us to work unconsciously toward many of our objectives.

Thus, in a product development organization, even someone whose job is limited to interface development is working toward many goals simultaneously, just one of which is developing a good interface. Many interface developers have professional responsibilities that also include system design, programming, marketing, managing, and so on, creating even greater parallel goal-driven activity. Common parallel project goals include designing and implementing reliable software and getting products out on time and within budget. Equally evident, but of a different nature, are such goals as communicating design ideas, initiating promising projects, and assigning people where resources are needed. Other goals, such as insuring that people feel good about their work and are rewarded appropriately, may seem tangential to interface design. However, the web of activity in an organization is tightly woven; as the hundreds of people in an organization work skillfully, often unconsciously, toward thousands of goals, a decision made in one area to address one concern will inevitably have effects, however subtle, elsewhere. Indeed, steps taken to reach virtually any goal could influence an interface design. These indirect effects may not be intended or even noticed; yet they can systematically distort the interface, introducing elements that work against the eventual users of the product.

This distortion is particularly likely in the absence of knowledge about what would be better for users. Confronted with a set of interface design alternatives and lacking knowledge of what would best serve users, we may decide on one alternative because it helps us attain another goal as well. For example, one design may be chosen because it is easier to describe and communicate. In this

section of the article, I describe possible undesirable influences exerted on the interface by goals that are seemingly tangential to interface design. Developers would be well advised to watch for these conflicting forces, which often go unnoticed, and persist in efforts to learn about users and their work environments. The competing goals are often genuinely important, so a solid case is often needed to convince colleagues to compromise them on behalf of the users.

Following a brief description of well-known software development goals that trade off against one another and against interface optimization, I discuss other goals under the categories of individual goals, group or team goals, and organizational goals. These attributions are necessarily imprecise. An individual may internalize group or organizational objectives. It can be difficult to pin down the objectives of groups, those often amorphous and overlapping organizational units wherein individual and organizational aspirations merge. Complete consensus regarding corporate goals is not likely. Nevertheless, the existence of personal objectives is evident, and one finds conscious efforts to create solidarity around specific team and organizational objectives.

Another complication is that the same behavior often serves multiple goals, including goals operating at different levels. For example, the goal of design simplicity may reflect a designer's personal aesthetic values or contribute to the team's engineering process or both.

This list is not comprehensive. It represents conflicts that have been observed to occur in some organizations. It should therefore be considered suggestive and intended to portray some of the complexity of the development environment. The hope is that readers will be vigilant lest these or other goals operate quietly to distort the interface design process.

4.1. SOFTWARE DEVELOPMENT GOALS

Certain constraints that can affect the interface are particularly familiar because they force tradeoffs or compromises in other aspects of software development. These competing goals are often the greatest impediment to usability. Because most developers are all too aware of them, I mention them and then focus on more subtle problems that often escape notice. The relative lack of attention by no means suggests relative lack of importance.

The goals of minimizing memory and processor use. The declining cost of computer memory and processing time makes better interfaces possible and creates a demand for better interfaces by making computation accessible to more people. Nevertheless, minimizing the use of these resources is still a major goal of most projects. This creates pressure, for example, to constrain the size of on-line help or reference texts and to limit processor-intensive interface activities. Research systems that explore advanced interface modalities, such as video, and advanced techniques, such as user modeling and coaching, utilize resources that are still not available to most products.

The goal of modular code. A major barrier to rapid advancement in interface design is that a hallmark of good software design—modular code organization—works directly against a hallmark of useful user support—knowledge of context. The effectiveness of the help, advice, or tutoring that a system can provide is directly related to its ability to judge a user's context at that moment. Ultimately that can require reconstructing the very information that a well-designed system or application has carefully buried in a dispersed set of pointers.

The goals of producing reliable, maintainable, secure software. Reliability and maintenance can be adversely affected by the complexity of interfaces flexible enough to accommodate people with differing roles, experience, and preferences. Security considerations can impose a human memory load and require computer users to carry out extra steps.

These and other goals integral to software development, some of them described below from more specific perspectives, can compete with usability. They are all valid goals in their own right. In the absence of information about what computer users really require, it can be very difficult to justify interface design approaches or choices that compromise the efforts to reach these goals.

4.2. COGNITIVE PROCESSES AND INDIVIDUAL GOALS

System development gives rise to many goals that operate primarily at the level of the individual developer. In addition to goals that originate in system development responsibilities, individuals have personal goals arising from career plans, preferences, style, and even personality. Conflicts based on these goals can emerge in the turbulent world of software development in general and interface development in particular. Again, influences from different levels flow together. Personal attitudes shape the interactions between programmers and members of

other groups contributing to the design and development of the interface; the organizational context also plays a major role.

System development is a highly rational activity. Cognitive activity is central to much of it, whether occurring at a terminal, with pen and paper, or in group settings. Of course, other processes are involved: perceptual and motor at the terminal; motivational, emotional, and social in group contexts; and organizational in still broader contexts. In considering individual goals in the workplace, however, cognitive processes are central. Behavior is governed by cognitive abilities and limitations interacting with tasks and objectives. The manifold goals present in the development environment act through cognitive tendencies to influence interface design. A few of these cognitive tendencies are discussed next.

Our inability to forget. We may often complain about our bad memories, but at times it would be convenient to be able to forget on command. If we could put out of mind what we know about how a computer works or how we ourselves use it, we could better imagine how someone less knowledgeable or with a different job will experience it. But this we cannot do.

Selective memory for salient events. Our memory is not uniform. We remember some things better than others. This can interfere with objective design; we exaggerate the importance or frequency of the things that we attend to and ignore the frequency or detail of unexceptional activity that takes place.

Difficulty making conscious the unconscious. We are not even aware of most activity—our own and that around us. We are so highly skilled and practiced at acting to reach many of our goals that doing so does not require conscious reflection. The resulting lack of awareness makes it difficult to detect or deduce the true sources of many design decisions.

Difficulty consciously handling multiple "channels." Although practice helps, we are not good at consciously working out the interactions of complex, multi-layered situations. Because this describes most work situations (in both user organizations and in software product development environments as outlined here), we are not well equipped to understand these interactions. With practice we may learn to handle such complexity unconsciously, but in delegating the process to uninspectable intuition, we allow other highly practiced activities to become part of the basis for our actions. The effects on design of "other highly

practiced activities" and the nondesign goals that they serve are explored in this section of the article.

Individual Goals Arising From Professional Responsibilities

The Goal of Understanding the Software Architecture

Apart from the effects of carelessness, neglect, and ignorance,[6] the most common source of poor interface design is probably the natural tendency of developers to map elements of the underlying software architecture onto the interface. This is useful for developers: If the interface reflects the underlying design, only one model of the system must be kept in mind. But users arrive with a model that does not correspond to the software architecture, so they are forced to acquire a second model of the system. In the early days of a new technology, most users are technically proficient, and it is both convenient and useful to reflect the architectural or engineering model in the interface. But a less technical user population needs different interfaces. The engineer sees relations among functions based on their implementation, whereas users construct different relations among the same functions based on their roles in the users' tasks.

Gentner and Grudin (1990) describe several cases in which the engineering model of a system was imposed on the interface to its detriment. For example, one commercially available VCR remote control labelled its buttons 0-9 and A-F. This corresponded to the underlying hexadecimal representation but confused most users, who prefer labels such as *Fast Forward* and *Reverse*. In another case, a software development team resisted adding a new function to a menu even though users expected it to be there, because the function was implemented quite differently from the other functions on that menu. Similarly, developers decided the *Print* function should handle all system objects consistently according to their internal structure, even when this led to printing a folder (i.e., a directory) index when users expected to have the objects in the folder printed (Grudin, 1989).

Here is a final example: A user trying a new application opened a sequence of windows in the course of carrying out an action. When this person closed one of the windows, all of the windows disappeared, one at a time. The user reported,

---

[6] These effects are (perhaps leniently) regarded here as unsystematic and are not discussed further. If detected through prototype testing or some other formal or informal means, the ill effects of carelessness or oversight do not draw support by serving other goals and thus may be relatively easily addressed.

> I expressed my surprise to (a programmer familiar with the code, who) responded immediately that the cascading window closing made perfect sense, based on the C-code cascaded procedure calls, with each newly called procedure opening a new window. [The programmer explained further, then] looked at me expectantly, waiting for the light of comprehension on my face. Instead I said that my new knowledge of how procedure calls caused the cascaded closing didn't make me like the cascaded closing... But this person didn't get it.

Even when intellectually recognizing the need to separate the software design from the interface design, an engineer cannot willfully forget the architectural model and assume the users' perspective. This point was emphasized by Gould and Lewis (1985) and illustrated metaphorically by Landauer (1988) with a "hidden figure" in a photograph: It is difficult to see the figure, but once it has been pointed out, you will always see it. You cannot return to the naive state.

The Goal of Design Consistency

"Experimental results... show that consistency (leads) to large positive transfer effects, that is, reductions in training time ranging from 100% to 300%" (Polson, 1988, p. 59). "Build consistent user interfaces" (Rubinstein & Hersh, 1984, p. 220). "Strive for consistency. This principle is the most frequently violated one, and yet the easiest one to repair and avoid [violating]" (Shneiderman, 1987, p. 61). "The common application of design rules by all designers working on a system should result in a more consistent user interface design. And the single objective on which experts agree is design consistency" (S. L. Smith & Mosier, 1986, p. 10).

Developers have indeed taken the goal of consistency to heart, and it is true that unmotivated inconsistency is likely to cause users problems. Unfortunately, especially for those who seek formal approaches to generating or monitoring for consistency, consistency can work *against* good interface design in several ways. As was noted, an interface that is consistent with the underlying software architecture may not be ideal for the computer user, such as when menu items are grouped according to aspects of their implementation rather than considerations of their use. The use of engineering terms, such as the BREAK key, can be convenient and consistent for engineers but not for other users (for example, one corporate vice president attended a demonstration of IBM's Presentation Manager,

which defines how the OS/2 system displays information, expecting it to be about slide or transparency preparation and viewing). Similarly, it can be consistent but undesirable to reflect an engineering perspective in error or status messages, such as “read error” when “check for disk damage” would be more helpful.

Another problem is that consistency with a real-world analog can facilitate ease of learning while obstructing ease of use, or vice versa. For example, several calculators initially adopted alphabetically arranged keyboards. This arrangement helps in some initial learning situations due to its consistency with our experience of the alphabet, but it slows down experienced users. The optimal keyboard design is not consistent in any obvious way at all. (Other examples are described in Grudin, 1989.) Especially when we are relatively ignorant about the users’ work practices, consistency is a helpful guideline, but considerations such as dialogue efficiency or the prevention of damaging accidents outweigh consistency in importance in many situations (Grudin & Norman, 1991).

Consistency as a goal in itself also obscures the problem of choosing dimensions on which consistency could be useful; there are often many conflicting possibilities, each is consistent, but not all of them are good. Everyone recognizes a foolish consistency, such as consistently capitalizing every other letter in command names, but subtler cases arise. Sometimes the proper choice depends on the circumstances. For example, in abbreviating command names, truncation (*de* for DELETE) is better when users will know the name and have to recall and type the abbreviation, whereas vowel deletion (*dlt* for DELETE) may be better when the users will see the abbreviation on a keycap and have to recall what it stands for, and single letters (*d* for DELETE) may be better for highly overlearned or prompted commands for which typing economy is critical (as in electronic mail options). A developer could create a consistent set of abbreviations, but if it is mismatched to the users’ tasks, it will be a poor design.

Only by understanding the users’ situations can developers get a handle on this matter. Consider the task of categorizing wildflowers. Botanists sort on the basis of leaf and petal shape, and many nature guides reflect this organization. But amateur nature lovers may start from the color of the flower, and some wildflower books organize flowers in sections by color. Insects may categorize flowers by smell and deer by taste. Since they can’t read, we don’t find books organized in these ways, but the example illustrates the diversity of possible

approaches to objects. Consistency chosen with the wrong task in mind can work against good design.

The Goal of Design Simplicity

In a review of work on programming aesthetics, Leventhal (1988) explored what it is about certain programs that appeals to programmers. Among the factors frequently mentioned was the notion of simplicity. "Programs have a kind of simplicity and symmetry if they're just right," said one programmer (p. 527). The author summarized that "themes of simplicity and wholeness, as well as the taming of complexity are suggested" (p. 530).

Simplicity serves other goals as well. A simple design is likely to be easier to communicate to other team members, to test, and to modify. For all of these reasons, a simple, elegant technical design is highly prized.

One might assume that computer users will find a simple interface easier to use. This is not always true. Unmotivated complexity is likely to cause problems, but when people already make or would benefit from making a particular distinction, the simpler interface created by eliminating the distinction will be an inferior interface. For example, business letters and computer programs can both reside on a system as ASCII files and even be edited by the same editor, but it is useful to use different iconic representations to distinguish them.

Grudin (1986) described a problem that arose in the design of a desktop system in which, under the surface, all documents are located in a single central catalog. A user sees a document displayed in a folder (i.e., directory) when, internally to the system, a pointer to the address of the document appears in the directory. A document is *shared*, appearing in two directories, when identical pointers are placed in each directory. From the system perspective, the two directories share the document on an equal footing; they cannot be treated differently because the pointers are indistinguishable. However, users may expect the original document—the one in the folder where it was first created—to be treated differently than the linked version created later. For example, a REMOVE command could result in different dialogues for original and linked copies. This would require more complex pointers that reflect information about object creation. But the developers preferred to stick with the simple, elegant pointer implementation even when shown that the resulting interface dialogue could not accommodate users' expectations. In a similar case, McHenry, Lynch, and

Goodman (1990) describe an information system consisting of thousands of files, each containing text notes entered by researchers over several years. The creator of each text note was recorded, so it was considered an unnecessary complexity to record the original creator of a file. This was fine at first, but the lack of a nominal owner eventually caused problems as users came and went; new users hesitated to extend or modify existing files, because they were unsure whom to ask about the files' purpose or use.

Another desktop metaphor example illustrates the tradeoff between simplicity and a potentially useful complexity. Many systems represent all directories (collections of documents or other files) by folder icons. Some systems add complexity by also providing cabinet icons and cabinet drawer icons. Like the folder icons, these correspond to directories, essentially indistinguishable to the system (for example, one can place a cabinet inside a folder just as easily as the other way around). Does having three different representations for directories complicate things? The complexity is introduced to provide a methodical way to organize directories hierarchically. Strictly to help users manage their files, the designers added a distinction that is not required by the system. Having used both systems, I feel that in this case the designers wisely overcame the attraction of simplicity (although opinions on this matter differ).

D. C. Smith, Irby, Kimball, Verplank, and Harslem (1982) described the eight main goals pursued by the designers of the Xerox Star. Two of these were consistency and simplicity. But the authors recognized the limitations of these goals and concluded that behavioral testing of a system is critical because even good designers' intuitions are error-prone and "simplicity, like consistency, is not a clear-cut principle. What is simplest along any one dimension (e.g., number of buttons) is not necessarily conceptually simplest for users."

Interface simplicity is not inherently bad; quite the contrary, many simple designs are good. But simplicity is a different goal than goodness (usability and utility), and the two should not be confused. Shneiderman (1987, p. 3) quoted Nelson on the goal of simplicity:

> Designing an object to be simple and clear takes at least twice as long as the usual way. It requires concentration at the outset on how a clear and simple system would work... It also requires relentless pursuit of that simplicity even

> when obstacles appear which would seem to stand in the way of that simplicity.

Sometimes, one of the obstacles to simplicity is good interface design.

The Goal of Anticipating Low-Frequency Events

A system or application design must cover all contingencies. Because extremely low-frequency situations are precisely those that may escape diagnostic testing, developers think carefully about possible combinations of conditions and events. Discovering a potential problem through logical analysis earns the admiration of one's peers. For the same reason, diagnostic testing itself must focus attention on low-frequency events.

This concern with complex contingencies, coupled with our tendency to exaggerate the salience of the things we focus on, gives extreme cases a disproportionate influence on the design. Interfaces may highlight features for circumventing or recovering from rare problems, creating clutter or distraction. For example, security procedures that burden users not only appear in particularly sensitive information systems but also can complicate interfaces in environments where mischief is highly unlikely.

Some features provide little benefit while increasing the size of the software interface, documentation, and training, and the testing and maintenance burdens. Someone can no doubt devise a scenario in which such a feature would be useful, so its elimination can require empirical evidence of the subtle harmful consequences and the infrequency of the positive contribution. Alternatively, a potentially useful feature that will break down in exceedingly rare situations may be abandoned early in the design process, when it could have been retained with suitable notifications or safeguards. Preoccupied with logically possible contingencies that can become very convoluted, we forget how rarely they will occur. The engineering maxim, "if something can go wrong, it will," must be kept in perspective. Too often the baby is thrown out with the bath water.

Some interfaces do a good job of moving rarely used features to low-level menus or bringing them out in appropriately timed messages or prompts, rather than forcing them on our attention. Some developers use empirical data to improve interfaces by eliminating such features (Gould et al., 1987). However, this aspect of design must be monitored particularly carefully because several

other goals described below also lead to undue emphasis on low-frequency operations.

The Goal of Thoroughness

Every software procedure goes through a similar process of being described, written, reviewed, and documented. A standard documentation practice may require one procedure per page. These and other checklist activities reinforce a perspective in which all functions are of uniform significance. This goal of being methodical and thorough, while an important part of the development process, disguises the tremendous variability in importance and frequency of use of different operations to users. The goal of thoroughness thus conceals the frequency and importance of some features in the computer use environment, just as does the goal of anticipating low-frequency events.

This can distort the interface in several ways. The HELP function on some systems lists dozens of commands in alphabetical order, confronting users with options that are obscure, ambiguous, and difficult to differentiate. Their natural importance to a user should be reflected by considering task-based categories and frequency of use.

Similar confusion or distraction results from printed documentation that lists functions alphabetically or lists one per page in a manner that reflects the development documentation on which it is based (and is more amenable to automatically generated documentation). If one knows nothing about the computer use environment in which the application will be used, this may be the best one can do. But any information about that environment, such as the potential frequency of use or probability of co-occurrence in a task, should make possible a more useful arrangement.

Unfortunately, much experimental work in human-computer communication has ignored the variability in feature use. For example, most studies of menu organization that have examined menu depth, breadth, and other variables test all menu items uniformly. Yet we know that menu items are not selected uniformly in work settings. Variations in frequency of use may outweigh most other factors in producing an optimal menu organization.

Individual Goals Arising From Personal and Career Issues

The Goal of Protecting Turf (Retaining Responsibilities)

Specialization in interface design is a relatively recent but rapidly spreading phenomenon in product development companies. In the past, programmers and software engineers developing interactive software typically designed the interface along with the other components. Many still do, but the growing importance of improving interfaces creates roles for interface specialists. Specialists include human factors engineers, technical writers, graphic designers, and training developers. Specialization will increase as new media—color, sound, video, animation—are more widely incorporated. Some argue for the involvement of specialists in the dramatic arts (e.g., McKendree & Mateer, 1991; Mountford, 1989).

This new division of responsibility may be resisted by engineers who are losing part of their traditional responsibility (e.g., Blomberg, 1988). Some engineers are happy to be relieved of this uncertain part of the design, but others see the interface as an enjoyable challenge or are acutely aware that users will judge the software on the basis of the interface—the visible part of the product.

The Goal of Staying Current (Extending Skill Repertoire)

In addition to retaining existing responsibilities, some developers see opportunities to broaden their contribution by working with newly emerging interface elements and modalities. They find it interesting to work in these new areas or feel that they must, lest their skills become obsolete. Engineers with no prior experience in graphic design, assigned to a project centered on a bit-mapped display, may enjoy designing icons, window borders, and so on. An engineer whose graphic designs are acclaimed within the programming group may resist working with (or yielding to) a real graphic artist whose expertise would improve the product.

The goals of retaining and extending one's responsibilities, although potentially undermining interface quality, are consistent with one of Gould's (1988) four key principles for designing usable systems: Keep responsibility for usability under one roof. When one programmer has total responsibility, this condition is met. Although that is not Gould's intent, it nevertheless points to a fundamental organizational challenge presented by the explosion of

specialization: How to coordinate the specialists effectively. This key problem, which will recur in discussions of goals at the group and organizational levels, is particularly strongly felt in areas where programmers feel particularly competent, such as technical writing (Grudin & Poltrock, 1989).[7]

The Goal of Personal Expression

People have important, largely unconscious, goals of achieving psychological balance by obtaining attention, approval, respect, power, and so on. This is often unstated in the engineering context, which prizes the antithetical goal of highly rational behavior. Nevertheless, psychological factors are critical to the success of development projects (e.g., Boehm, 1988; Shneiderman, 1980). Actions taken in furtherance of these personal goals have clear influences on system development, including interface development. The "not invented here syndrome" as a barrier to adopting ideas is an example.

Actions undertaken to satisfy personal goals can undermine the cooperation between software engineers and interface specialists in support roles: human factors, technical writing, industrial design, training, marketing, and so on. Nelson (1990), a prominent developer and writer, felt strongly about this problem:

> Historical accident has kept programmers in control of a field in which most of them have no aptitude: the artistic integration of the mechanisms they work with (in interface design). It is nice that engineers and programmers and software executives have found a new form of creativity in which to find a sense of personal fulfillment. It is just unfortunate that they have to inflict the results on users (p. 243).

---

[7] Two cases in which development teams excluded writers are described by Good (1985) and Gould et al. (1987). These studies indirectly illustrate the danger of such exclusion. In the former, technical writers handled the printed documentation, but a software engineer designed the help and error message text. Of the 362 suggestions received during iterative development, the largest category was "suggestions for improving the wording of particular help or error messages that users found to be confusing" (p. 573). In the latter study, engineers retained control of the user guide, and "we iterated over 200 times on the English version of the user guide" (p. 762). Each case study is presented as a demonstration of iterative design. Iteration is unquestionably extremely valuable, yet one can ask whether *such extensive* iteration would have been required had professional writers designed the help messages, error messages, and user guide.

4.3. SOCIAL PROCESSES AND GROUP OR TEAM GOALS

Group dynamics introduce social, motivational, emotional, and political concerns that are less apparent when focusing on the work individuals do in isolation. Some group- or team-level goals are unstated, possibly even unrecognized, and thus their influences are less apparent. The key activities of communication and coordination are largely cognitive and amenable to rational analysis, whereas goals such as equitable treatment of group members and cooperation among them introduce explicitly motivational and emotional elements.

The Goal of Communicating a Design

Communication is a fundamental activity in large product development organizations; communication skills vary, but they are heavily exercised. Because few interface designs in such organizations are written, reviewed, approved, and implemented by one person, they must be communicated. A design that is easily communicated helps meet this goal. Unfortunately, the resulting interface may not end up being so easy to use.

For communicating a design, paper has been the preferred medium and brevity a preferred style. Some designs are easy to describe on paper, and some are messy. A set of pull-down menus that are no more than two levels deep, with each menu containing about ten items, looks elegant on the printed page. In contrast, a design with branch points that extend three or four levels deep in places, containing some areas bristling with options and others sparsely populated, looks messy. In the absence of hard evidence as to which users will prefer, why not go with the easily communicated design? It seems to make sense that an easily described design will be easy to use.

But this is often untrue. A user's dialogue with the computer is narrowly focused and extends over time, in contrast to the static, spatially distributed written design. Presenting options at different points as a user proceeds can make good sense, as can menus that vary significantly in length and depth, depending on the contents, despite an inelegant or confusing appearance on paper.

Paper presentation may disguise interface problems that users will stumble over repeatedly. Consider an entire set of pull-down menus displayed on one page. Readers searching for a particular item can find it in seconds, even if it is not under the most obvious heading. But actual users never see the entire set of

menus simultaneously. If an item is under the wrong heading, they may wander around, inspecting possible synonyms or dropping down to search lower menu levels fruitlessly. They may do this several times before finally learning the location of the item.

Similarly, designers may avoid placing an option in two places in the interface. It looks inelegant on paper and the gratitude felt by a user wandering down the avenues of the interface is difficult to imagine. (The tendency to map the software architecture onto the interface and the "checklist approach" noted above also work against including multiple paths to a single operation. Procedures and their definitions appear only once in the code and the code documentation.)

Problems for users originating in decisions that serve the goal of communication among developers are hard to detect on paper, yet they can be uncovered quickly with simple prototypes tested in conditions approximating work settings. It has been suggested that prototypes could supplement or even replace documentation for communicating design proposals. This is worth exploring, but prototypes, too, communicate some things better than others.[8]

The Goal of Coordinating a Development Project

Conway (1968) observed that design architectures often reflect the organizational structure of the development organization producing them. He concluded a clever analysis with the observation,

> To the extent that an organization is not completely flexible in its communication structure, that organization will stamp out an image of itself in every design it produces. The larger an organization is, the less flexibility it has and the more pronounced is the phenomenon. (p. 30)

The interface design is no exception to this rule.

As noted, perhaps the most prevalent systematic source of interface problems is the imposition by developers of the "engineering model of the system" (i.e., the

---

[8] Even superficial aspects of communicating on paper can influence design decisions. A professional-looking design document impresses people. In one development group in the mid-1980s, the Macintosh quickly became an indispensable design tool when design documents incorporating its graphics outshined competing design documents.

underlying software architecture) on the interface. This allows developers to work with a single model that covers both the system and the interface. The "management model of the project" represents an analogous situation. Managers must assign implementers to functions, and they also encounter these functions in reviewing the interface design. (If the team follows the often-recommended practice of writing "a user manual" first, the manager deals with both simultaneously.) It greatly simplifies things when the models coincide: when the interface reflects the assignment of programming tasks.

A manager can find it easier to make the case for an additional programmer to code a specific function if the function is called a *utility* (and ultimately appears on the UTILITIES menu) than if it is described as being part of a function that is being coded by someone else. However, users might expect to find the two functions together, rather than one moved to a UTILITIES menu. Similarly, when one implementer handles several functions, the functions may gravitate to the same interface location. And once aspects of the interface have been assigned to different people, integrating them in the interface can require an investment in additional communication and coordination.

Many default design decisions made for the sake of convenience in this manner can be quickly reversed if they are shown to cause problems for users. But that evidence must be provided.

The Goal of Compensating Developers

It is desirable to reward a designer who creates a novel feature or a programmer who completes a difficult implementation. One way to do so is to increase the work's visibility through prominent placement in the interface. From a user's perspective, however, the appropriate time and place to provide access to a feature may be buried in an interface that gives prominent placement to more ordinary features. When the goal of compensation takes precedence, the result is that a new command is defined when it would be better to add a parameter to an existing command, a new function key is assigned instead of extending the use of an existing function key, and so forth. To appreciate this phenomenon, imagine the difference between being able to tell friends "I created one of the UNIX

system commands," and "I coded an option you can select by setting a flag on one of the Unix system commands."[9]

It can require solid evidence of user work patterns or preferences to justify a decision to bury a feature in an interface. A friend once joked, "I work on this 2 years and you get to it by hitting CONTROL X in some other application?" Other goals can work in concert with the goal of compensation to promote visibility at the expense of usability. Visibility can also serve marketing purposes: It may appeal to buyers or customers, not users. (Sometimes a customer subsequently becomes a user; sometimes the customer acts on behalf of users and focuses exclusively on functionality.) Visible placement in the interface can also be the result of a new feature's prominence in the developers' minds.

Several of the forces described above can coalesce into an irresistible case for a design,without any consideration of the users' work requirements. When developers collectively define the interface, as is often the case, the developer of a given feature sees it as distinct from the work done by others, understands its unique implementation, and regards equal prominence in the interface as a reasonable reflection of the effort it required; in addition, the result makes sense to management.

Consider an example: Several stand-alone, menu-driven office systems developed in the early 1980s include a spelling checker that appears high in the menu hierarchy. To use it, one closes the document being edited, moves up in the menu hierarchy, selects the spelling checker, then reenters the name of the document that was just closed. To resume editing after spelling correction, one accesses the word processor and reenters the document name. This is extremely inconvenient: One would generally prefer to enter the spelling checker from within the word processor (although perhaps not by typing CONTROL X!). Whatever factors contributed to this decision—ease of implementation, marketing visibility, the perceived importance of the feature, the need to coordinate or compensate developers, or others—the decision was governed by goals other than usability![10]

---

[9] The intention is not to pick on Unix, which may have reduced early excesses of this sort. For example, the old DSW command has become the "-i" option in the RM command.

[10] In Polson and Lewis (1990), this word processor interface is examined by a program that formally analyzes interfaces for usability problems. The program detects syntax problems in

The Goal of Cooperation

Compromise and finding workable tradeoffs are characteristic of work in development environments. To improve the interface requires sacrifices elsewhere. It would be easier if the interface reflected the underlying software architecture. It would be easier if the interface reflected the organization of the code documentation. It would be easier if formal properties of consistency or simplicity could be applied to interface design. Where is the tradeoff point, when do we compromise? If we rely on intuition and have no firm evidence of usability problems, being cooperative can mean compromising our intuitions concerning the interface all too often.

Substantial give and take is present in an engineering environment that is ideally driven by rational decision-making. People "call in IOUs" and "cash in a lot of chips" to reach important goals. As mentioned in Section 3, the lack of feedback from existing users robs the interface of something that would give it additional weight: the awareness that it affects a lot of people for a long time. Because developers can quickly put it out of mind, the interface is easily compromised. Even when the users' perspective is in focus, it can fall victim to good-natured compromise.

Several years ago, developers from two of a company's merging product lines met to unify their "look and feel." One product used a 24-line display and presented status messages on the first line. The other product used a 25-line display with messages on line 25, at the bottom. Future customers would use both products together, so consistency was desirable. Hours of debate over the relative merits and the difficulty of changing each product led to no agreement. Finally, a solution was proposed: The two products would display prompts consistently—on line 25. On the 24-line display, this would of course wrap around to the first line. The proposal was accepted.

To avoid unduly sacrificing the interface in the spirit of cooperation, the voice of the users must be heard. Developers have difficulty arguing with users; we have all heard the expression "the customer is always right." An

---

command parameters but did not catch this feature access problem. To find this problem requires an understanding of the users' work environment that is beyond the grasp of such systems. While formal interface analysis can add to a developer's inventory of tools, it will not replace the need for user involvement.

oversimplification, perhaps. But we never hear "the user interface specialist is always right." So a user is a good ally to have.

4.4. ORGANIZATIONAL PROCESSES AND CORPORATE GOALS

An organization can be considered a unit that interacts with other organizations, reacts to external events, and develops internal structures and processes in support of these activities. Organizations work to prevail (or at least to survive) and to develop corporate cultures that provide purpose and continuity. Organizations can create as well as react to their environments (Mintzberg, 1984). Mintzberg argued that only certain combinations of internal and external conditions lead to organizational stability, which suggests that organizations should adopt specific sets of goals in order to reach, maintain, or shift among stable configurations. The concept of organizational goals simplifies a reality in which conflicting goals are held by different managerial groups, but this observation only reenforces the view of an organization as a cross-current of goal-driven activity. This section is limited to exploring the influence of a few basic organizational goals on interface design.

The Goal of Efficient Division of Labor

In Section 3, I described the effects of division of labor in separating developers and computer users. Contact with customers and users is channeled into sales, marketing, field service, customer support, training, and management. Expanding this picture, Figure 1 shows that the groups responsible for different parts of the interface are dispersed throughout an organization. Those in marketing define high-level functionality, software and human factors engineers in development design the low-level software dialogue, the technical publications department writes the documentation, and a training group develops on-line training and training courses. Translations and other adaptations for international versions of a product may be done in different countries. Communication occurs via written requirements (e.g., between marketing and development), functional specifications (supplied to technical writers by development), and the actual software (provided to training developers). Opportunities for miscommunication and lack of coordination are legion.

This dispersal of responsibility extends to software itself. As noted earlier, developers often do not know the application set with which their application will

be marketed and used; if they do, the developers working on the other applications (or even on other parts of their own) may not be accessible. As a result, users encounter confusing differences in the naming, styling, and organization of functions. Establishing a "corporate look and feel" is a partial solution, but it risks placing a "foolish consistency" ahead of good design as an interface goal.

The general neglect of on-line help is an example of divided responsibility affecting interface design. A good help system might save a company a substantial expenditure on customer hand-holding, service calls, and printed documentation. These savings would be in the budget for customer service. But the effort and expense of developing the help system would come from the development department; this group is rewarded primarily for producing new products and functionality. Affirmative action to promote on-line help is especially necessary given a possible lack of developer empathy with less experienced users, a condition that is reenforced by the lack of mutual contact. Thus, help systems often end up with a low development priority.

Division of responsibility is necessary, and existing approaches efficiently serve many organizational functions. But the current organizational structures worked more efficiently for the development and marketing of noninteractive systems in an era of less complex user requirements. The lack of experience that companies have coordinating interactive systems development increases the difficulty of working across organizational boundaries. Mutual education and trust are inhibited by differences in outlook and even language: Sources of confusion include differences in the way certain terms are understood by developers, on the one hand, and customers and marketing representatives, on the other (Grudin, 1991a).

How are these problems addressed? Matrix management is one approach to overcoming organizational separation. Employees from different groups with a range of skills are temporarily assigned to a project. However, due to the perception or the reality that the contribution of technical support roles is limited to certain phases of a project, such assignments are often of limited duration and effectiveness. The language barrier remains and is accompanied by other problems, such as an ambiguity and difficulty in evaluating work. An example of a matrix management effort succumbing to these forces is described in Grønbæk, Grudin, Bødker, and Bannon (in press).

Development projects sometimes try to absorb the technical support functions (human factors, technical writing, training development, etc.) by assigning them to programmers or by hiring individuals directly for a project. This may work for a time, but when a project is completed and the product is turned over to others for revision or maintenance, responsibility for the interface elements designed by the internal groups (screen layout, documentation, training, etc.) reverts to the relevant external support groups in the organization. Sensitive about its initial exclusion, an external group may cheerfully throw away the version developed by the team and start over. The resulting confusion does not benefit users.

The increased need to understand user requirements has led to beefing up the use of internal and external mediators: marketing and systems analysts, consultants, and so on. Is this more gradual approach an adequate response to the challenge? Those calling for direct user involvement in development are explicitly questioning its soundness. This issue is addressed in Section 5.

The Goal of Managing Development

Development projects must be managed, and the management process inevitably intrudes on many aspects of the activity. The intent is that the intrusions be beneficial on the whole and of minimal harm in each area. But there is growing recognition that widely advocated software development methods systematically impede the development of usable interactive products. This came about due to the environments in which these methods originated.

Most computer processing was done in batch mode; interactive systems were rare. An emphasis on careful early design makes sense for noninteractive software, with its relatively predictable development course. It works less well for the interface, where uncertainty about a workable design is the rule. This is the motivation for prototyping and iterative design with user involvement. Gould and Lewis (1985) emphasized the distinction of interactive software:

> "Getting it right the first time" plays a very different role in software design which does not involve user interfaces than it does in user interface design. This may explain, in part, the reluctance of designers to relinquish it as a fundamental aim. In the design of a compiler module, for example, the exact behavior of the code is or should be open to rational analysis... Good design in this context is highly analytic, and emphasizes careful planning. Designers

> know this. Adding a human interface to the system disrupts this picture fundamentally. A coprocessor of largely unpredictable behavior (i.e., a human user) has been added, and the system's algorithms have to mesh with it. There is no data sheet on this coprocessor, so one is forced to abandon the idea that one can design one's algorithms from first principles. An empirical approach is essential. (p. 305)

Computing resources were too expensive to devote much to early interfaces. For example, the cost of memory once precluded extensive help texts or even error messages, and processor time was too expensive to permit users to examine help texts on-line anyway. Those constraints have been disappearing quickly, but development processes and priorities shift more slowly.

Contract and internal development projects were the focus, not product development. The multi-stage waterfall model of software development arose in the context of large government projects. Competitively bid contract development naturally emphasizes written specifications; a Catch-22 element is that developer contact with the eventual users is often prohibited or discouraged after requirements have been defined, which occurs prior to selecting the developers. Separate contracts may be awarded for system design, development, and maintenance. The resulting development approaches include structured analysis, wherein the task "establish man-machine interface" is relegated to one sub-phase of system development (De Marco, 1978), and Jackson System Development, which "excludes the whole area of human engineering in such matters as dialog design" (Jackson, 1983, pp. ix-x). Variants of the waterfall method were widely promulgated; when computer companies turned to product development it was natural to try to apply it. Of course, because such methods do not specify on-going user involvement in design, project plans do not anticipate it.

Most computer users were engineers who understood or were happy to learn the system. As noted under the goal of understanding the software architecture, when the computer use environments are similar to a computer development environment, the interface that emerges during development and debugging is often adequate or even appropriate for use. A legacy of that era is the widespread belief that the interface can be ignored or tidied up at the end of development. This was perhaps true for interfaces to engineers and programmers, but is not true for interfaces serving increasingly diverse user populations. The lack of

information about user environments is a source of inertia: The degree to which development and use environments have diverged is not appreciated.

In summary, the methods in use today originated to support the internal and contract development of expensive batch systems that were handled by engineers and trained operators, conditions that strongly discouraged user involvement in development and that do not describe contemporary product development contexts. New approaches to development built around prototyping and iterative design (and thus incorporating user involvement) are emerging (e.g., Boehm, 1988; Perlman, 1989), but they have yet to be proven or widely adopted. Unfortunately, as noted earlier, the high visibility and growing importance of the interface works against iterative design in three ways: (i) the interface is grouped with aspects of the product that must be "signed off" on early in development; (ii) other groups, such as those producing documentation, training, and marketing, are strongly tied to the software interface and affected by changes; and (iii) iteration or change in the interface is noticed by everyone, which can create uneasiness, especially in an environment with a history of stressing early design.

Solutions to these problems can and will be found but will require changing the way we work. Unfortunately, an innovative process proposal is unlikely to find management as receptive as is a detailed product design specification.

The Goal of Effective Decision-Making

Product development organizations rely on the informed intuitions of individual managers. These intuitions are prone to special lapses when applied to a range of interface decisions.

Decision-making involves a tradeoff between two desirable goals: maximizing the soundness of a decision and reaching the decision in a timely, efficient manner. Inevitably, trained intuition plays an important role. Executives and development managers generally have good track records, but interface development is a new concern: Decision-makers have not had the experience to build or demonstrate their intuition in this area.

The visibility of an interface brings it to management's attention. One developer cited in Poltrock (1989c) commented,

> If you put up an architecture model, not many people will come back and comment... But if you actually put up screens and ask people to comment,

> everyone from the managing director down has their own personal idea... The person who has designed the user interface sits there in a meeting and gets bludgeoned by a person 17 levels higher than them who just says, "I want my feature to behave like this." (pp. 10-11)

The only solution proposed to this common problem is to obtain solid evidence from users for given design alternatives (Gould et al., 1987).

Decision-makers in development environments often feel they know what they would like as users. But they may not be typical users or even computer users at all; they all too often support interface technologies, such as voice and natural language, that promise to reduce learning time to zero but that have remarkably long histories of exaggerated forecasts and failure (Aucella, 1987; Grudin, 1988, 1990; Johnson, 1985).[11] The problem is two-fold: (i) the technologies themselves are extremely difficult to perfect (or even render adequate)—projects in these areas have been characterized as "black holes" (Williams, 1990)—and (ii) when available in circumscribed form, the technologies succeed only in restricted niches—for people who make heavy, direct use of computers, which excludes most decision-makers, the drawbacks usually outweigh any advantages that the current state of the art provides.[12]

The failure of intuition is a particular problem in developing groupware, products designed to support groups (Grudin, in press). Someone with good intuition may be able to use a spreadsheet for an hour and decide that many users will like it, but no one person's intuitions can be expected to cover the range of

---

[11] Actually, many managers would like computers to be like people, with whom they interact skillfully. More explicit than most, Nicholas Negroponte, director of the MIT Media Laboratory, says "computers should be more like people," (1990b, p. 246) and "the computer must be an old friend" (1990a, p. 351).

[12] This problem has an impressive pedigree. Licklider and Clark (1962) included speech recognition and natural language understanding that could handle syntactic, semantic, and pragmatic aspects of language among 10 prerequisites for true human-computer "symbiosis," although they were more aware of the difficulty than many who followed. Exaggerated predictions dogged the 1970s and 1980s. In "Information Technologies for the 1990s," Straub and Wetherbe (1989) forecast that "human interface technologies" will be the information technologies with the greatest impact, and speech recognition and natural language will be the key human interface technologies. Not surprisingly, the article was based on interviews with nine corporate chairmen, presidents, and executive directors, and one business school professor. Equally optimistic is Negroponte's August 1989 prediction that "the most significant development in the human/computer interface during the next five years will be in speech technology" ("Speech More Important," 1989, p. 26).

needs of group members who differ in role, experience, and preferences. Not surprisingly, development managers tend to favor interfaces and applications that promise to benefit managers—project management applications, decision support systems, meeting scheduling and facilitation systems, voice applications, natural language interfaces, and so on—not realizing that these usually require other users to do extra work, sharply reducing the likelihood of a successful product. In turn, the developers of such applications focus on interface features that support the principal beneficiary, a manager, neglecting or impeding the interfaces to other, often more frequent, users.

The Goal of Competing in the Marketplace

In addition to the inward-looking concerns emphasized thus far, large product development organizations manage external relationships with customers, competitors, and others. These concerns create powerful forces that often intrude directly into the design process. One, mentioned earlier, is the strategic need to conceal marketing plans, with the result that information about the eventual users is concealed from developers.

Another force is the pressure for frequent releases that creates the time pressure mentioned in Section 3. As competition and the pace of change increase, product development companies are pressured to turn out enhancements and new products in a rapid, predictable fashion. In reading the following analysis, consider the implications for interface development in general and user involvement in particular:

> Ashton-Tate's decline began with what is becoming a well-worn story in the industry: failure to upgrade a market-leading product. Dbase III Plus went for almost three years before being upgraded, while competitor's products were upgraded as often as twice in that time. (Mace, 1990, p. 43)

A similar pattern of predictable new releases is found in other maturing markets, from automobiles to stereo systems. The result is pressure for a predictable and controllable software development process: for routinization of development. Parker (1990) described a proposed solution to the problem described in the previous quotation:

> Lyons [an Ashton-Tate executive] responds that he can keep customers by providing predictable if not always exciting upgrades. "Customers don't want to be embarrassed; they want their investment to be protected. If you are coming out with regular releases, even if they skip a release because a particular feature is missing, they will stay [with the product] because the cost of change is large." (p. 44)

This strongly felt need for rapid, controlled development creates difficulties for design elements or approaches that have uncertain duration or outcome. Interface design generally has a relatively high level of uncertainty, and user involvement can increase development time and introduce the possibility of changes in direction. This is the intent, of course—a better design means a changed design—but it nevertheless works against the powerful pressures for predictability.[13]

An external force that is particularly potent when technology and the marketplace are changing rapidly is the installed base. Existing users are simultaneously a blessing and a curse. They provide a relatively reliable market for new products, but exert a conservative pressure against change. Yet, change must be introduced: The last remaining users of paper tape and punch cards may like their familiar technology, but product lines are eventually retired. Interface change in particular requires existing users to adjust. Resistance to an interface change can be overcome by determining the cost of adjustment; that is, by measuring the degree of disruption and the eventual productivity gain, data that are rarely collected.

Unfortunately, demonstrating that most or all users will benefit from a change is not always enough. Even worse than having a less than state-of-the-art product is having a reputation for abandoning existing customers. However small or irrational the group opposing change, the product development company wants to avoid being known for deserting its users. New companies can innovate without running this risk. But established companies eventually have outmoded features

---

[13] This pressure can mold the development organization and process. In particular, there is a trend to routinize development—to eliminate dependence on any one person. Arguments that discount an increased deskilling of programming have focused on the internal software development centers of large corporations (see Friedman, 1989), not on software product development; the latter is where the competitive pressures that motivate increased control are most evident.

cluttering product interfaces and development resources channeled into extending their lives at the expense of innovation and greater usability.

A company that introduces and promotes a new feature can obtain a marketing edge, whether or not the feature provides any real benefit for users. In response, competitors will develop the same feature without ascertaining or even caring about its contribution to usability. Thus, we see the spread of unused software features.

Finally, the goal of interface optimization is not always supreme, particularly if optimization is defined in terms of the performances of individual users and groups in isolation from a broader context. The acceptance of a standard represents a decision to arrest optimization in order to accrue other benefits. For example, the QWERTY typewriter keyboard layout is known to be suboptimal but it is deemed to be good enough. As software applications mature, formal and de facto standards freeze significant aspects of interface development. New concerns about interface copyright violation may also discourage incremental improvement in favor of licensing existing technology.

5. WAYS TO PROCEED

This section describes approaches that are being used or could be used by product development companies to increase developers' understanding of users and their work. Different approaches are likely to be appropriate or possible in different circumstances. I begin the section by examining positive conditions for obtaining direct user involvement in product development projects, then discuss approaches for overcoming obstacles to achieving direct user involvement, and finally examine alternatives to direct contact with users.

5.1. Positive Conditions for User Involvement

To one working within a large product development organization, the obstacles sometimes seem insurmountable. But there are grounds for optimism: Product development companies also provide support for involving users in interface design, primarily by putting interfaces themselves in the spotlight and providing incentives to improve them.

Ease of learning and ease of use become important marketing edges as software products mature. Adding a new bell or whistle does not help much if the already available functionality is underutilized. A better interface is one way to

distinguish a product and to increase its acceptance in a competitive marketplace. Applications are reaching out to discretionary users, people who have the choice of whether or not to use a computer, and the greater availability of alternatives further increases buyer discretion. Computer users are likely to consider usability in exercising discretion.

Large product development organizations often have considerable resources (development costs are highly amortized), and declining hardware and software costs permit more computational power to be directed to the interface. Human factors engineers and interface specialists employed by these companies are in the forefront of research and development in this field.

Relatively frequent upgrades and product replacements can help developers break out of "single-cycle" development. Evaluation of existing product use can feed into the design of later versions, and good ideas arriving too late for use on a specific project can be retained for later use (Grudin, Ehrlich & Shriner, 1987). In addition, projects to develop upgrades or replacements have users of the existing versions as good candidates for involvement in development.

In contrast with in-house projects, product development efforts usually have a large supply of potential users, and the fate of a product is not so heavily dependent on situational factors operating in a given site. Developers may interact with potential users without inadvertently jeopardizing the project by offending them or raising unduly high expectations. In contrast with some competitively bid contract development situations, product developers encounter few legal constraints on interacting with users.

Finally, large software development companies with established product lines may become resistant to change, but they were founded on change and recognize at a deep level that they must change to survive. This leads to some inherent openness to experimentation, which can be amplified by evidence that the forces that work systematically against user involvement in development stand in the way of product optimization and success.

5.2. Processes That Incorporate User Involvement

In seeking to overcome the obstacles, the power of persistence should not be underestimated. Usability and contextual engineering approaches have been refined and used in product development (Gould, 1988, and Whiteside, Bennett,

& Holtzblatt, 1988 are excellent summaries). Prominent, successful case studies include the development of the Xerox Star interface (Bewley et al., 1983; D. C. Smith et al., 1982) and the IBM Olympic Message System (Gould et al., 1987); less heralded successes are numerous. All have found ways to involve users, generally in iterative prototype definition and evaluation.

Another promising resource is the experience derived from other development contexts, notably European projects based on internal or in-house development. Partly because some of the obstacles described in this article are less salient in such contexts, user involvement is more often achieved. Progress on developing the issues, techniques, and tools in this area are described in Bjerknes, Ehn, and Kyng (1987); Floyd, Mehl, Reisin, Schmidt, and Wolf (1989); and Greenbaum and Kyng (1991). Of course, adapting what has been learned in internal development to product development is a significant challenge.

In the UTOPIA project (an acronym in the Scandinavian languages for "training, technology, and products from a quality-of-work perspective), described in Ehn, 1989, some of these approaches were explicitly applied to a product development effort. A small number of potential users—newspaper typographers—were heavily involved with the developers, and techniques including a newsletter were used to involve a much broader selection of potential system users on a more limited basis.

Experiments with prototype testing and iterative development are increasing our understanding of when and how they are most effectively used. Boehm's (1988) spiral model of development builds these techniques into a disciplined software engineering methodology. He is one of several writers encouraging an explicit change from the current focus on defining the development *product* to a focus on defining the development *process*. Grønbæk et al. (in press) described a project that succeeded only after this shift occurred in midcourse.

Due to the growing demand for more usable systems, practitioners may find a climate for limited experimentation with these and other approaches. But even to begin working effectively requires a clear awareness of the obstacles, an understanding of why they are there, and a tolerant recognition that their source is in institutional constructs, not in unsympathetic individuals.

5.3. Technology to Support User Involvement

Rapid prototyping tools, a long anticipated solution, are finally becoming widely available. A natural outgrowth of the participatory design efforts is a focus on prototyping tools that facilitate real-time modifications as computer users and developers work together (Grønbæk, 1990). Tools that provide a smoother transition from a prototype to a production application will reduce the problems encountered with prototypes that do not readily scale up (Glushko, in press). Just as code management systems have proven to be useful, single-user rapid-prototyping tools could benefit from features to support the collaborative nature of most development.

Video and other multimedia software support will enable development project members to communicate more effectively across distances. Equally important, video is potentially a powerful tool for communicating users' experiences to developers. It can effectively convey both the specific details and the general richness of work environments.

The need to bridge the information gap between development and use environments is so great that the computer should come to play a direct role. Before information about computer use can be communicated directly to developers, issues of privacy and confidentiality must be worked out; in some environments even benign monitoring will not be possible. But so great are the potential advantages, for computer developers and for computer users, that efforts to find a mutually satisfactory arrangement will be amply rewarded.

5.4. Strengthening the Use of Mediators

As we gain experience with interfaces and as their importance increases, the traditional mediators or indirect communication paths between developers and users are improving. Systems analysts, marketing personnel, consultants, user groups, and others are becoming more skilled at identifying and describing interface needs. Standards organizations devote more attention to interface issues. Managers, software engineers, human factors engineers, and others who participate in development projects are becoming more knowledgeable about interfaces and interface development methods. Trade publications, trade shows, journals, conferences, and books disseminate more information each year. Also positioned between developers and the end users are other development and sales organizations, some of which are described in the next section.

The remarkable proliferation of these mediators may be a response to the relative difficulty of establishing direct communication between users and developers. The success of these mediators in capturing and conveying the necessary information varies. Clearly, they are likely to be more reliable in mature application domains than in new areas.

5.5. Redefining the User Population

As computer use extends to more application domains and as product maturity increases, large product development companies must decide who their customers are. They can focus on specific end user markets or can channel their efforts into building platforms for third-party developers and value-added resellers who have experience in specific domains. Both trends were noted by the president of Philips, the Dutch electronics company, when he announced plans to narrow its focus to specific market segments, such as banking. "There is a shift from selling directly to customers and retailers to selling indirectly through value-added concerns who tailor the products to the customer needs" ("Philips, Facing Losses," 1990, p. 13). The independent value-added resellers, in turn, prefer to work on industry-standard software platforms.

In taking this path, software is following hardware. Long ago, hardware ceased being sold directly to end users without a layer of system software. Now, system software and a standard application set is insufficient: More customers require domain-specific software as a condition of purchase.

This mediating process is illustrated by a case in which an internal group in a large company is developing software for telephone operators who take equipment orders. The software will run on a workstation being developed by a major computer company. In constructing and testing prototypes, the internal development group has achieved a high level of user involvement, working closely with several operators. At the same time, these developers are serving as potential future users in a study conducted by members of the product company's workstation development team. Thus, the group is participating in two projects, one as developer and one as user; each development group has identified its own user population.

The fundamental need for knowledge about users remains unchanged, but the identity of the users has changed. When the users are software developers who customize or extend a product for specific markets, the gap to be bridged is

different and perhaps shorter. The product development company has in effect delegated responsibility for understanding the ultimate users of the system to the internal developers, value-added resellers, and other intermediaries operating in specific domains. As these mediating groups grow in size and number, they are likely to encounter many of the same challenges that the large product development organizations have faced.

What must product developers know about interfaces in this scenario? They should know which features their developer-customers will want to tailor, and the corresponding ranges. Interface features that will be passed directly on to end users require as much attention as ever.

5.6. Redefining the Development Company

In the long term, organizational structures and development processes may evolve, institutionalizing solutions to the problems described here. However, there is little sign of this happening quickly; many of the approaches that have been described in this section are being applied largely within existing structures.

Traditional software methods are under pressure due to the recognition that they are inadequate for developing interactive systems (e.g., Boehm, 1988). Methods such as prototyping and iterative development are widely accepted as being necessary. Many design faults that result from carelessness or haste, as well as many that originate in conflicting goals in the workplace, can be relatively easily filtered out with such techniques. But, as was noted, current organizational structures and practices work against the application of these methods. In addition, their application requires active user involvement. Bringing this about will require organizational change, beginning with a greater awareness of the challenge.

Most organizations would have to be restructured to follow Gould's (1988) key principle of putting all aspects of usability under a single management. Gould's injunction to be prepared to work in a "sea of changes" describes a situation more suited to an adhocracy than to the typical bureaucracy (Mintzberg, 1984). Although older companies may have difficulty changing, companies that formed as the present conditions were emerging may show more of these qualities. And Gould's vision is realized in the younger, smaller, third-party developers, value-added resellers, consulting, and other companies that move in to

insulate the large companies from end users. They may be the interface laboratories of the future.

6. CONCLUSION

The software development process is one of constant compromise. Tradeoffs are forced by conflicting engineering considerations, management decisions, and product marketing constraints. Development time is pitted against additional functionality; hardware capability vies with price considerations. Tradeoffs also emerge from the other forces described: One design is easier to communicate than another, one group deserves greater recognition, two projects compete for one available engineer, and so forth. Designing and building a quality interface is always a goal, but it is just one of many. In the absence of a strong case for particular choices, aspects of the interface are undervalued when the tradeoffs are resolved. Interface quality is compromised much too readily.

In Section 3, I outlined obstacles that product developers face in improving interfaces by involving users in design and evaluation. In Section 4, I described competing goals that underlie those obstacles or that act directly to distort interfaces. In most cases, activity in furtherance of these goals is so highly practiced that side effects on interface design go unnoticed. The best way to overcome these forces—or more accurately, to balance them, since they represent legitimate goals—is to obtain clear evidence of what is required to enhance usability. In the struggle to satisfy multiple constraints, silent, weak, or indecisive users' voices are not a strong constraint.

A secondary purpose of this article is to hint at the complexity of workplaces: to show what will be involved in understanding work environments well enough to position computers in them. The workplaces described here are special—product development environments—but it is reasonable to assume that computer use environments, although often differing radically from development environments, are equally complex. Developing sufficient understanding of their complexity will not be easy.

Product development companies can address the problem of increasing developers' understanding of users and their work in several ways, described in Section 5. They can reorganize or restructure their development methods to increase direct contact, use technology to increase the flow of information, increase their use of mediators, and redefine their user population.

ACKNOWLEDGMENTS

Tom Malone provided crucial encouragement in 1985. Through 5 years of collecting data and organizing observations, Susan Ehrlich Rudman and Steve Poltrock were invaluable colleagues. Tom Dayton, Michael Good, John Gould, Gary Perlman and John Richards helped with specific issues. Wang Laboratories and MCC provided opportunities to experience and study these phenomena, and Aarhus University provided the time to put this together. The paper was improved considerably by comments from Phil Barnard, Elizabeth Dykstra, Tom Erickson, Don Gentner, Bob Glushko, Michael Good, Tom Malone, and an anonymous reviewer, and by particularly careful reviews by John Bennett and Judy Olson.

REFERENCES

Aucella, A. F. (Moderator). (1987). Voice: technology searching for communication needs. *Proceedings of the CHI+GI'87 Conference on Human Factors in Computing Systems,* 41-44. New York: ACM.

Bewley, W. L., Roberts, T. L., Schroit, D., & Verplank, W. L. (1983). Human factors testing in the design of Xerox's 8010 'Star' office workstation. *Proceedings of the CHI'83 Conference on Human Factors in Computing Systems,* 72-77. New York: ACM.

Bjerknes, G., Ehn, P., & Kyng, M. (Eds.). (1987). *Computers and democracy: A Scandinavian challenge.* Brookfield, VT: Gower.

Blomberg, J. (1988). The variable impact of computer technologies on the organization of work activities. In I. Greif (Ed.), *Computer-supported cooperative work: A book of readings* (pp. 771-781). San Mateo,CA: Kaufmann.

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

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

De Marco, T. (1978). *Structured analysis and system specification.* New York: Yourdon.

Ehn, P. (1989). *Work oriented design of computer artifacts.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Floyd, C., Mehl, W.-M., Reisin, F.-M., Schmidt, G., & Wolf, G. (1989). Out of Scandinavia: Alternative approaches to software design and system development. *Human-Computer Interaction, 4,* 253-349.

Friedman, A. L. (1989). *Computer systems development: History, organization and implementation.* Chichester, UK: Wiley.

Gentner, D. R., & Grudin, J. (1990). Why good engineers (sometimes) create bad interfaces. *Proceedings of the CHI'90 Conference on Human Factors in Computing Systems*, 277-282. New York: ACM.

Glushko, R. J. (in press). Seven ways to make a hypertext project fail. *Technical Communication, 38,* 3.

Good, M. (1985). The iterative design of a new text editor. *Proceedings of the Human Factors Society 29th Annual Meeting,* 571-574. Santa Monica, CA: Human Factors Society.

Gould, J. D. (1988). How to design usable systems. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 757-789). Amsterdam: North-Holland.

Gould, J. D., Boies, S. J., Levy, S., Richards, J. T., & Schoonard, J. (1987). The 1984 Olympic Message System: A test of behavioral principles of system design. *Communications of the ACM, 30,* 758-769.

Gould, J. D., & Lewis, C. H. (1983). Designing for usability—key principles and what designers think. *Proceedings of the CHI'83 Conference on Human Factors in Computing Systems,* 50-53. New York: ACM.

Gould, J. D., & Lewis, C. (1985). Designing for usability: Key principles and what designers think. *Communications of the ACM, 28,* 300-311.

Greenbaum, J., & Kyng, M. (Eds.). (1991). *Design at work: Cooperative design of computer systems.* Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Grudin, J. (1986). Designing in the dark: Logics that compete with the user. *Proceedings of the CHI'86 Conference on Human Factors in Computing Systems,* 281-284. New York: ACM.

Grudin, J. (1988). Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces. *Proceedings of the CSCW'88 Conference on Computer-Supported Cooperative Work,* 85-93. New York: ACM. Revised as Why groupware applications fail: Problems in design and evaluation, *Office: Technology and People, 4,* 3, 1989, 245-264.

Grudin, J. (1989). The case against user interface consistency. *Communications of the ACM, 32,* 1164-1173.

Grudin, J. (1990). Groupware and cooperative work: Problems and prospects. In B. Laurel (Ed.), *The art of human-computer interface design* (pp. 171-185). Reading, MA: Addison-Wesley.

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

Grudin, J. (1991b). Obstacles to user involvement in software product development, with implications for CSCW. *International Journal of Man-Machine Studies, 34,* 435-452.

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

Grudin, J., Ehrlich, S. F., & Shriner, R. (1987). Positioning human factors in the user interface development chain. *Proceedings of the CHI+GI'87 Conference on Human Factors in Computing Systems,* 125-131. New York: ACM.

Grudin, J., & Norman, D. A. (1991). Language evolution and human-computer interaction. *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Grudin, J., & Poltrock, S. (1989). User interface design in large corporations: Communication and coordination across disciplines. *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems,* 197-203. New York: ACM.

Grønbæk, K. (1990). Supporting active user involvement in prototyping. *Scandinavian Journal of Information Systems, 2,* 3-24.

Grønbæk, K., Grudin, J., Bødker, S., & Bannon, L. (in press). Achieving cooperative system design: Shifting from a product to a process focus. In D. Schuler and A. Namioka (Eds.), *Participatory design*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Jackson, M. (1983). *System development.* Englewood Cliffs, NJ: Prentice-Hall.

Johnson, T. (1985). *Natural language computing: The commercial applications.* London: Ovum.

Landauer, T. K. (1988). Research methods in human-computer interaction. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 905-928). Amsterdam: North-Holland.

Leventhal, L. M. (1988). Experience of programming beauty: some patterns of programming experience. *International Journal of Man-Machine Studies, 28,* 525-550.

Licklider, J. C. R., & Clark, W. E. (1962). On-line man-computer communication. *AFIPS Conference Proceedings 21,* 113-128.

Mace, S. (1990, January 8). Defending the Dbase turf. *InfoWorld,* pp. 43-46.

McHenry, W. K., Lynch, K., & Goodman, S. E. (1990). Task, group, system: The collaborative research "package." Unpublished manuscript.

McKendree, J., & Mateer, J. (1991). Film techniques applied to the design and use of interfaces, *Proceedings of the 24th Hawaii International Conference on System Sciences, Vol. 2,* 32-41. Los Alamitos, CA: IEEE Computer Society.

Mintzberg, H. (1984). A typology of organizational structure. In D. Miller and P. H. Friesen (Eds.), *Organizations: A quantum view* (pp. 68-86). Englewood Cliffs, N.J.: Prentice-Hall.

Mitch Kapor face to face. (1987, January). *INC. Magazine,* pp. 31-38.

Mountford, J. (Moderator). (1989). Drama and personality in user interface design. *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems,* 105-108. New York: ACM.

Negroponte, N. (1990a). Hospital corners. In B. Laurel (Ed.), *The art of human-computer interface design* (pp. 347-353). Reading, MA: Addison-Wesley.

Negroponte, N. (1990b). The noticeable difference. In B. Laurel (Ed.), *The art of human-computer interface design* (pp. 245-246). Reading, MA: Addison-Wesley.

Nelson, T. H. (1990). The right way to think about software design. In B. Laurel (Ed.), *The art of human-computer interface design* (pp. 235-243). Reading, MA: Addison-Wesley.

Parker, R. (1990, January 8). Bill Lyons' task: Incremental moves to consistency. *InfoWorld,* p. 44.

Perlman, G. (1989). Asynchronous design/evaluation methods for hypertext technology development. *Hypertext'89 Proceedings,* 61-81. New York: ACM.

Philips, facing losses, to trim 10,000 jobs. (1990, July 3). *International Herald Tribune,* pp. 1, 13.

Polson, P. (1988). The consequences of consistent and inconsistent user interfaces. In R. Guindon (Ed.), *Cognitive science and its applications for human-computer interaction* (pp. 59-108). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Polson, P. G., & Lewis, C. H. (1990). Theory-based design for easily learned interfaces. *Human-Computer Interaction, 5,* 191-220.

Poltrock, S. E. (1989a). Innovation in user interface development: Obstacles and opportunities. *Proceedings of the CHI'89 Conference on Human Factors in Computing Systems,* 191-195. New York: ACM.

Poltrock, S. E. (1989b). *Participant-observer studies of user interface design and development* (MCC Tech. Rep. No. ACT-HI-125-89). Austin, TX: MCC.

Poltrock, S. E. (1989c). *Participant-observer studies of user interface design and development: Communication and coordination* (MCC Tech. Rep. No. ACT-HI-162-89). Austin, TX: MCC.

Rubinstein, R., & Hersh, H. (1984). *The human factor*. Bedford, MA: Digital.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems.* Cambridge, MA: Winthrop.

Shneiderman, B. (1987). Designing the user interface: Strategies for effective human-computer interaction. Reading, MA: Addison-Wesley.

Smith, D. C., Irby, C., Kimball, R., Verplank, B., & Harslem, E. (1982, April). Designing the Star user interface. *Byte,* pp. 242-282.

Smith, S. L., & Mosier, J. N. (1986). *Guidelines for designing user interface software* (Report No. 7 MTR-10090, Esd-Tr-86-278). Bedford, MA: MITRE Corporation.

Speech more important interface than graphics, Media Lab's Negroponte tells SIGGRAPH. (1989, November). *Byte,* p. 26.

Straub, D. W., & Wetherbe, J. C. (1989). Information technologies for the 1990s: An organizational impact perspective. *Communications of the ACM, 32,* 1328-1339.

Whiteside, J., Bennett J., & Holtzblatt, K. (1988). Usability engineering: our experience and evolution. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 791-817). Amsterdam: North-Holland.

Williams, M. (1990). Panel statement, summarized in: The loyal opposition. *Proceedings of the CHI'90 Conference on Human Factors in Computing Systems,* 54. New York: ACM.