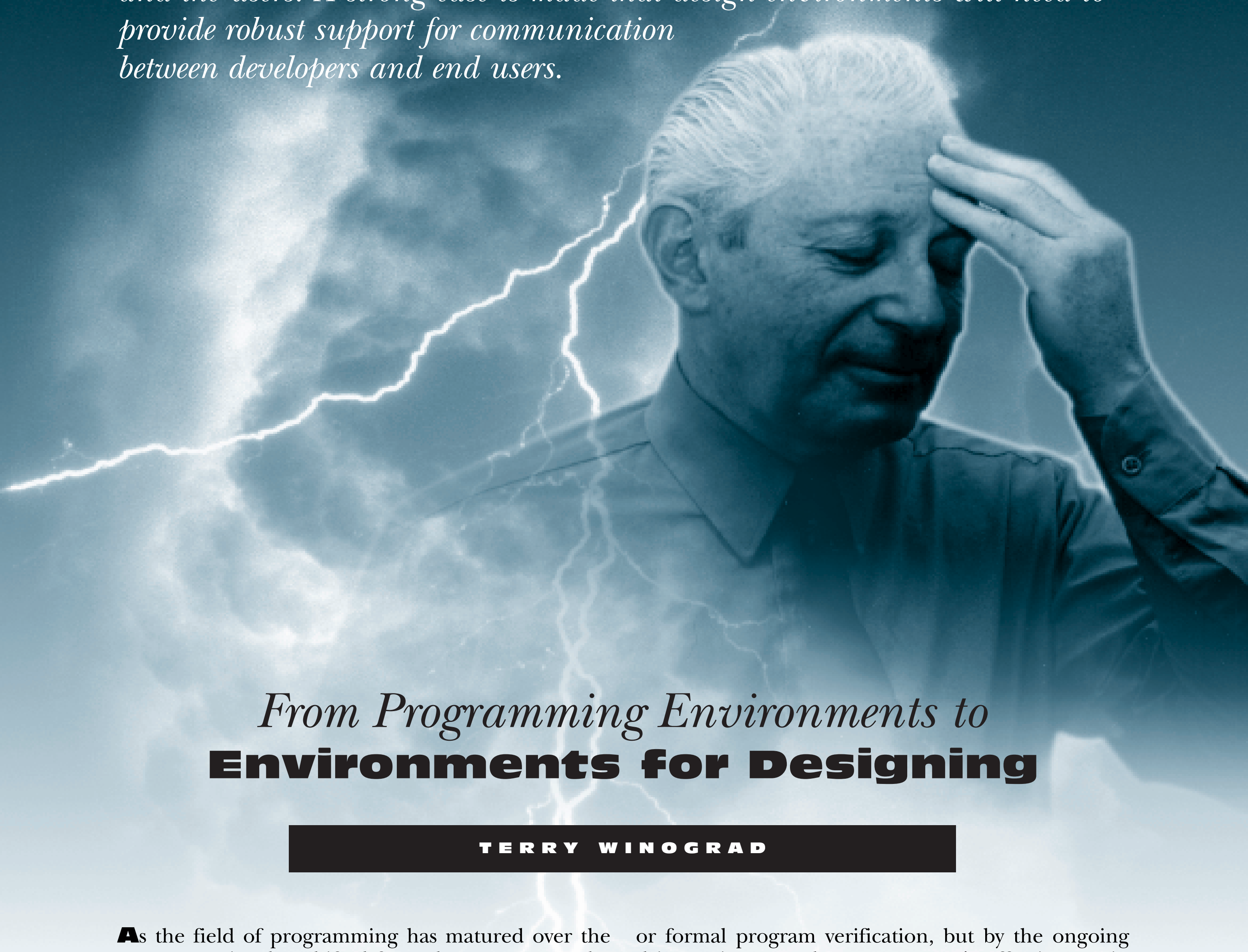
Though there have been advances in end-user programming,[*] complex applications still need professional developers. This inspired look at the future of creating complex software explores the shift from programming environments to design environments, discussing environments that help developers satisfy end-users' cognitive needs and help deal with contextual issues such as the aesthetic, practical, and social properties of the application and the users. A strong case is made that design environments will need to provide robust support for communication between developers and end users.



From Programming Environments to **Environments for Designing**

TERRY WINOGRAD

As the field of programming has matured over the years, attention has shifted from the program to the programmer—from the logical and computational structure of algorithms to the cognitive structures of the people who produce them. Innovations such as interactive programming environments, object-oriented programming, and visual programming have not been driven by considerations of algorithm efficiency or formal program verification, but by the ongoing drive to increase the programmer's effectiveness in understanding, generating, and modifying code.

This special section of *Communications* clearly reflects this shift of emphasis from machines to people, exploring the cognitive aspects of programming languages and environments. This article moves to broaden the view still further: from programming to design. It does not focus on cognition or on programming, but on the wider picture of software design and the larger environment, which

*See B. Nardi's *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, 1993; and *Watch What I Do: Programming By Demonstration*, by A. Cypher, Ed., MIT Press, 1993.

goes beyond the standard tools of the programmer's trade.

The first part of this article portrays the evolving field of software design, and its relationships to the traditions of programming, analysis, and design that have served the computer field throughout its history.

The second part relates the concerns that are highlighted in software design to traditional approaches in programming and programming environments. It draws analogies between four specific aspects of current programming environments and four corresponding aspects of environments for software design. As with most analogies, the results are intended to be suggestive, not rigorous. Starting with the cognitive aspects of well-understood programming tools, we can get insights into the demands for an environment that will support the ongoing development of computer software in a rapidly changing industry.

Technological Maturity

In his widely read *Software Design Manifesto* a few years ago [8], Mitchell Kapor bemoaned the fact that "Despite the enormous outward success of personal computers, the daily experience of using computers far too often is still fraught with difficulty, pain, and barriers for most people....The lack of usability of software and poor design of programs is the secret shame of the industry."

These are strong words to throw into the face of a multi-billion dollar industry that by all standard measures must be doing things right. But Kapor is highly respected as the founder of Lotus Software and the designer of Lotus 1-2-3, the "killer app" that gave a major impetus to the whole microcomputer industry. His concerns have resonance among many people who work with software. Although the unprecedented power of computing systems makes them highly useful, there is a big gap between what we see in most products today and what could be done to make them really usable. This becomes more pressing as we begin to reach beyond the current applications to new audiences and new ways of taking advantage of computation in people's lives.

In the last few decades the computing profession has matured from its early days, when ingenuity was required at every turn to make programs work at all. Today we are part of a major industry in which the expectations for successful programming constantly increase. Software designers today have the opportunity (and necessity) of moving to a broader view of what they need to achieve, because of the tremendous successes of computing.

We are now entering a new phase of computer product development, which can be understood as a step in a history of technological maturity that has been repeated for many new technologies, such as the radio, the automobile, and the telephone:

Phase 1) Technology-driven. In the first phase, a new technology is difficult to employ, its benefits are not yet obvious, and its appeal is mainly to those who are fascinated with it for its own sake—the "early adopters." We find clubs of enthusiasts who love to share stories about how they fought the difficulties and overcame them. The general public is seen as not having sufficient understanding or merit to really use the new inventions. Ham radio is a good example of a technology that was adopted by a small but dedicated group in the technology-driven stage. In the same vein, the legends of Silicon Valley include many stories of the early computer days and the brave pioneers who tackled the Altair or the Osborne.

Phase 2) Productivity-driven. In the second phase, the economic benefits of using a technology are developed to the point where people in industry and business will adopt it for practical uses. The measure is in the bottom line—not whether the technology is fascinating or easy to use, but whether it can be shown (or at least believed) to produce greater efficiency, productivity, and profits. The use of radios for truck and taxi dispatch, police, and military communication falls into this class, as do most of the major microcomputer applications sold today. Spreadsheets, word processors, databases, desktop publishing, and a host of other applications have been sold as tools to increase the productivity and competitiveness of companies that buy and use them. Design considerations are measured primarily in the realm of cost-effectiveness. If better design can speed up use, cut training time, or add to efficiency in another similar way, then it is important. If it cannot produce a measurable difference in one of these dimensions, then it is a "frill."

Phase 3) Appeal-driven. In the third phase, a maturing technology reaches a wide audience of "discretionary users" who choose it because it satisfies some need or urge. The emphasis is not on measurable cost/benefit analyses, but on whether it is likable, beautiful, satisfying, or exciting. The market attractiveness of a

Just what is software design? *How does it differ from programming, software engineering, software architecture, human factors, or any of the other labels that have been applied to the activities around creating computers and the programs with which people interact?*

In order to design software that really works we need to move from a constructor's-eye-view to a designer's-eye-view, **a view that takes the system, the users, and the situation of use all together as a starting point.**

product rests on a mixture of its functionality, its emotional appeal, fashion trends and individual engagement. CB radio and cellular phones for personal use are examples of radio technology in this third product phase. Computer games have been there since the beginning, and an increasing portion of computer use is shifting to the consumer end of the spectrum. The huge new markets of the future—the successors to the productivity markets conquered by IBM and Microsoft in the past—will be in this new consumer arena, responsive to different dimensions of human need.

Software design that focuses on the user, not on the mechanisms, is moving to center stage.

The Movement toward Software Design

"Software design" has become a slogan for the emerging shift of perspective, away from what the computer does, toward the experiences of the people who use it. From this perspective, the task of those who create new software is to design the interaction, not to design the program. Although the difference may be subtle (good applications programmers have always paid some attention to designing the interaction), many people are feeling the need to make the distinction by creating new professional identities and new affiliations. Some notable recent examples:

- In 1993, a professional organization named The Association for Software Design (ASD), was founded with the mission to "transform and elevate the status and quality of software design as an activity." It already has chapters in several locations around the U.S. and is initiating a program of educational activities.
- In 1994, a publication named *Interactions* was founded by the Association for Computing Machinery, in conjunction with the ASD and SIGCHI, the ACM special interest group on Computer-Human Interaction. In the inaugural issue, the editors stated, "We seem to have moved well beyond the idea that making a computer 'useful' is simply to design a good interface between 'man and machine.' Our ideas have evolved to the point where the richness of human experience comes to the foreground and computing sits in the background in the service of these experiences." [18].
- The entire first 1994 issue of the journal *Human Computer Interaction* (the primary academic journal in the field) was devoted to a dialog around an article by John Seely Brown and Paul Duguid on the role of context in design [3]. The editor said, "We can look at the development of the field of human-computer interaction as an evolution of what we in the field of HCI consider to be the significant aspects of context for computer-based artifacts." [11].
- At the CHI94 SIGCHI conference, Mitchell Kapor gave the keynote address, in which he argued for the primacy of design as an approach to HCI. The conference offered an unprecedented number of papers and sessions devoted to design issues. The 1995 conference instituted a new section called "design briefings," which are "specifically intended to provide increased exposure to user interface design and to practical user interface work. They involve the presentation of notable designs and a discussion of how these designs came to be."
- In August 1995 the first Symposium on Designing Interactive Systems will be sponsored by ACM SIGCHI, IEEE, and ASD. The call for participation says, "The time is ripe to address designing as a coherent activity—technical, cognitive, social, organizational, and cultural. The goal is to come to a better understanding of how designing works in practice and how we can improve it."

What is Software Design?

It is evident that software design is coming into prominence. But that should give us a moment's pause. Just what is software design? How does it differ from programming, software engineering, software architecture, human factors, or any of the other labels that have been applied to the activities around creating computers and the programs with which people interact? How does it relate to other fields that call themselves "design," such as industrial design, graphic design, urban design, and even fashion design? It is easy to make a new label. The real work is in creating a change in perspective that gives new directions and ideas.

The education of computer professionals has generally concentrated on understanding the nature of computational devices and the engineering that makes them behave as the builder intends. The focus is on the things being designed—the devices and programs and the parts that make them up. The goal is to fully implement a specified functionality in a manner that is robust, reliable, and efficient. When a software engineer says that a piece of software "works," he or she typically means that it is robust, reliable, and meets its functional specification. These concerns are indeed important. A designer ignores them at the risk of disaster.

But this inward-looking perspective with its emphasis on function and construction is one-sided. In order to design software that really works we need to move from a constructor's-eye-view to a designer's-eye-view, a view that takes the system, the users, and the situation of use all together as a starting point. When a designer says that something "works" (e.g., a book cover layout or a design for a housing complex) the sense is much broader —it works for people in a context of values and needs, to produce quality results and a satisfying experience. The key to this shift of perspective is in turning our attention to the larger context in which the object of design resides.

Traditional software engineering has dealt with context in an operational sense, relating a program to the operating systems, networks, programming interfaces and the like which will surround its operation. Software engineering techniques are geared to expand the possibilities for a program to be modified, ported to other systems, extended to new functionalities and adapted by users over a lifetime of use. But the focus is always on the mechanisms, not the human situations in which they will be embedded.

The perspective of software design shifts from the "outside-looking-in" focus on mechanisms to an "inside-looking-out" focus on people and their situations: how people experience software; what they do with it; and the larger situation in which they encounter it (see [23] for a number of current perspectives).

Environments for Software Design

The development of programming environments was an important step forward in software engineering. A good environment can embody and facilitate the principles and practices that make programming more productive. It brings the programmer's activities into focus along with the activity of the program being produced.

In an analogous way, we can better understand the user-oriented view of software design by looking at what might constitute a "software design environment." In a traditional programming environment, the objects of interest are programs, and the programmer's tools are designed to operate on various representations of those programs. The software design environment is concerned with designing the interactions, and works with a broader array of representations, including different kinds of conceptual models, mockups, scenarios, storyboards, and prototypes. The design methods reach outside of the workstation to include the setting and the thinking of the people who will use the software.

The activities of a software designer include the traditional activities of software engineering and programming, such as specification writing, coding, and debugging, along with the user-focused design activities emphasized in this article. Giving traditional programming concerns short shrift in the sections that follow does not imply that they are superfluous or that environments to facilitate them are unnecessary. The emphasis here is on developing our understanding of the additional activities that go on around and through them.

To highlight the software design perspective, four current topics of focus in programming environments will be examined. The analogous issues from the software design point of view are outlined in Table 1 and explained in the following sections. Of course, as with all analogies, there isn't a perfect fit, but the parallels can help elucidate the motivations and criteria for new design environments.

Responsive Prototyping Media

Interactive programming
Responsive prototyping media

Modern interactive programming environments emphasize quick turnaround—the ability of a programmer to try something out, see what it does, make changes, and try again in a tightly coupled cycle. In this activity, the nature of the programming language and environment makes a large difference—perhaps as large as the difference between sculpting in clay and sculpting in stone. The ability to quickly shape and reshape requires a capacity for turning an unarticulated idea into a working object quickly enough to be able to change it, listen to it, even throw it out and go on to another.

This kind of "reflective conversation with the materials" (see [19] for an excellent analysis of the nature of design activities, in which Schön introduces this term), is a key to effective design and is even more

Table 1. In expanding from programming environments to environments for design, there are suggestive correspondences between current techniques and what is needed for user-oriented software design.

Programming Environments	Environments for Software Design
Interactive programming	Responsive prototyping media
Specifications	User conceptual models
Reusable code	Design languages
Interactive debugging	Participatory design

important for the interaction-intensive programs that dominate today's software world. Both the interface and underlying functionality of the application are incrementally designed through interaction with the intended users. Both user and designer need to be able to visualize what the program will be like and what can be done with it, even before it is programmed.

Abstract representations, such as written descriptions, flow charts, and object class hierarchies cannot provide a grounded understanding. In the past few years, a number of techniques have been developed for initiating a dialogue with the user (and with designers) before writing program code, through mockups, storyboards, scenarios, and prototypes [10, 12]. In classical engineering practice, a prototype has been a kind of laboratory test, taking the concept for a device and demonstrating that a simplified version of it could be made to work. In current design practice, prototyping is primarily a vehicle for exploration and communication. Prototypes not only give feedback to the designers, but also serve as an essential medium for information, interaction, integration, and collaboration. The emphasis is on quickly providing an artifact that can be a concrete vehicle for letting the users (and the designers) see both possibilities and problems with the proposed design.

The key element isn't the accuracy or thoroughness of the prototype, but the communicative role it plays, both in the designer's interaction with the materials and the user's interaction with the designer. A traditional programming environment emphasizes getting the prototype to do the right things, a design perspective emphasizes getting it to communicate. A software design environment needs to support a variety of prototyping levels, suitable for different projects and different phases of a project, each making use of different tools.

Rough Hand Sketches and Scenarios. The initial step in visualization is to get something that has enough of the general look to suggest the functionality and interaction to those who see it and talk about it. This requires little in the way of technology. Poster board and marker pens may be all that are needed, and the relevant skills lie in being able to quickly sketch a rough vision, not a polished piece of art. By working with a sequence of sketches, a designer can explore a large number of possibilities for a program—not just its look, but its functionality. The sketches serve as a communication vehicle for users to envision new uses for a piece of software, and for giving insight into how that software will actually work in their situations.

Low-fidelity Prototypes (Wizard of Oz). Moving beyond static sketches, a number of techniques have been developed for giving the user a sense of the dynamics of a program without having to build a functional version of it. The simplest techniques can be implemented with paper technologies such as post-its and transparent overlays, manipulated by a human machine surrogate who can pop up and pull down menus, switch window contents, and so on. Even a rough attempt at duplicating the dynamics of the program being designed can give a surprising amount of new insight into what will work and what will falter.

The fact that these prototypes don't feel like a real product isn't a problem. In fact, in many design settings it is often important to make sure that prototypes at various levels have a feeling of roughness—even to the point of using scanned pencil sketches in place of more polished bitmap art. A user faced with something that has the feel of a rough sketch is more likely to respond with substantive suggestions. A designer is more likely to see strikingly different possibilities. A highly polished prototype—even if it is only a first attempt at the functionality and interface structure—fosters a sense of finality that tends to inspire only suggestions for minor improvements and further visual niceties.

Programmed Facades. "Potemkin village" prototypes can be built on the computer using prototyping tools such as Hypercard, Supercard, Macromind Director, and Toolbook. An interface produced in these languages can present a facade which appears on the surface to be a real program, and which may mimic some illustrative aspects of the functioning of the intended program. But this facade is often supported only by an illusion. The underlying logic of the prototype may duplicate only a tiny fraction of what is intended for the finished design, and it may work for only a carefully selected set of possible interaction sequences.

Even though the programmers know that much is missing underneath, the effect in communicating to others can be tremendous. They get a feel for the program that is impossible to get from looking at static screens, and they will be able to see many of its flaws and its new possibilities. In fact designers have sometimes found that by showing this kind of prototype to users or managers they create false expectations—it looks so good that it seems as if the real thing should be only a short step away. Usually there is a lot more to be designed!

Prototype-oriented Languages. There is no clear dividing line between facade-building languages and full-fledged programming languages that are designed to support the prototyping process (often at the expense of traditional computer language concerns, such as execution speed and economy of storage). Environments such as Hypercard, Smalltalk, and Visual Basic include interface builders that make it especially easy to design the screens that people will see and to attach working code to the visual elements.

Although these languages are sometimes thought of as the basis for throwaway demonstration prototypes, it has often turned out that for the specific intended use, a program written in one of these languages will be adequate for the job, and does not need to be reprogrammed into a "real" programming language. When deciding how much programming effort should go into a prototype, it is important for the designer to look at these trade-offs and see whether it should be thought of as a throwaway, or be written with the expectation that it could be the basis for the final implementation.

The full design environment is a mix-and-match of all of these prototyping levels. Some projects lend themselves only to one of them. Some projects will best use a mix of all. The goal is to use the level(s) that will best facilitate the two primary interactions—designer with design and user with designer.

Much effort has gone into the design of prototyping systems that can increase the software designer's fluidity of iterative design. This same kind of creativity acceleration is produced in other design disciplines in what Michael Schrage [20] calls a "culture of prototyping." He points out that different organizations develop and use prototypes in different ways. In some cases a prototype is a rough sketch to be passed around for quick comment and change. In others it is a carefully crafted selling aid, designed to get approval from a manager, customer, or committee that will decide on the future of a project. Some media (like the clay models used in automotive design) lead to resource-intensive prototypes in which a highly finished look leads to assessments of quality. Others, such as carved foam, lead to rough-cut prototypes whose visual and material qualities suggest their provisional status and openness to being changed.

As the field of software design develops, **we too are developing not one but many cultures of prototyping, and design environments to facilitate them.**

As the field of software design develops, we too are developing not one but many cultures of prototyping, and design environments to facilitate them. We are learning which of them is most appropriate to a given organization and task.

User Conceptual Models

Specifications

User conceptual models

A key element of many software engineering methodologies is the creation of abstract specifications. These characterize the desired system at a higher level than the operational code, and they can therefore be more easily understood, described, and manipulated. There are, of course, many controversies about the values and limitations of different specification formalisms and methodologies, but we will not address those here.

The analogy we want to draw to software design is with the "conceptual model" [14] or "virtuality" that lies behind the interface seen and manipulated by the user. One of the key differences between software and most other kinds of artifacts that people design is the freedom of the designer to produce a world of objects, properties, and actions that exist entirely within the created domain. The comprehensible but arbitrary consistency of a virtuality is most immediately evident in computer games, which gain tremendous appeal through the ability of the player to engage in the virtual world in earnest, exploring the vast reaches of space, fighting off the villains, finding the treasures, or dealing with whatever the designer creates. But there is also a world created in a desktop interface, a spreadsheet, or an information network. We are familiar today with the virtuality of the graphical user interface with its windows, icons, folders and the like. Although these are loosely grounded in analogies with objects in our everyday physical world, they exist in a unique world of their own, with its special physics and potential for action by the user.

The literature on interface design uses a number of terms for the world created by the software, such as Conceptual Model, Cognitive Model, User Data Model, User's Model, Interface Metaphor, User Illusion, Virtuality and Ontology. What these terms all share is the recognition that the designer and user are engaged in creating a world, not in simply bringing to the computer what existed outside of it.

In early computer program development, the virtual world was usually a side effect of the implementation. Users of Unix, for example, work in a world of files, directories, and links (symbolic and direct) because those were elements of the underlying system implementation. This direct mapping onto the implementation model works well for certain kinds of applications and certain kinds of users, but in general, the way of dividing up the world that works best for the computer is not the same as the one that will work well for human understanding and action.

In the development of the Xerox STAR in the early 1970s, designers began to directly confront the question of building a clear and understandable conceptual model [7]. Although the STAR did not have the commercial success of its later derivatives, it was the original model for consistent integration of now-familiar mechanisms for windows, icons, dialog boxes, drawings, and on-screen formatted text. Its interface innovations have been the basis for a whole generation of systems, including the Macintosh, Microsoft Windows, and Motif.

Rather than first deciding what the system would do, then figuring out how to produce interfaces, the developers engaged psychologists and designers from the beginning in an extensive set of storyboards, mockups, prototypes and user tests to see what would work, and how. In doing this, they recognized that the most important thing in designing properly was the users' conceptual model, and that everything else should be subordinated to making that model clear, obvious and substantial. Users could manipulate documents by moving and acting on the icons that appeared on the screen. But, of course, the icon isn't the document. The interface could just as well have used pinwheels or little text fragments, and could have let the user operate on them with different physical devices, commands, and visual effects. The key part of the design was the creation of a coherent and consistent world, or virtuality, with an understandable underlying structure or model.

Tools for designing virtualities have often been based on object-oriented models, in which the object classes reflect the user's perspective rather than being driven by implementation concerns. In fact, object-oriented programming began with the simulation language SIMULA, which started from the standpoint of representing and simulating real world objects. With later developments, such as Smalltalk and its descendants, designers realized that many of the objects they were creating did not reflect existence outside the computer, but had lives of their own in the virtual world with which users interacted. It is notable that current methodologies talk about object-oriented design rather than object-oriented programming. This is not to say that the only methods for conceptual design are object-oriented, but they have in common a concern with defining and describing the objects, properties, and operations that the user interacts with, rather than the algorithms or representations used by the computer.

Design Languages

Reusable code

Design languages

One of the major efforts in software engineering today is to find better tools for reusability. Object-oriented software, component software, linked libraries, and many other mechanisms are being explored to enable significant elements of a program to be rearranged and reused in others. In the design world, this kind of borrowing has always been standard, and is technically easier. The concept for a widget such as a tool bar, or an interaction style like a multiple selection can be copied from one application to another without technical difficulty.

A significant part of the larger design environment is the collection of design elements that have been previously used and are standard in a software culture. It doesn't take deep analysis to see that all of the current GUI interfaces draw on a basic vocabulary and interaction style that was pioneered in the STAR and then the Macintosh. In fact the great success of the Macintosh can be attributed to a large degree to the efforts of the early Apple "evangelists" to encourage applications developers to use a common design vocabulary. They facilitated this by publishing explicit guidelines [2], by providing tools for all of the standard elements (menus, dialog boxes, window management, etc.) and by working directly with developers. They convinced the developers that they would gain more from promoting the popularity of the Mac platform by making it seem easy to use through uniformity, rather than through having minor differences (improvements) unique to their interface.

The Role of Design Languages. The Macintosh was the first open platform to publicize a design language to use in designing software interaction (the STAR had a carefully articulated design language, but all of the applications were developed by Xerox). A number of design theorists have pointed out how the use of consistent and understandable language by the designer makes it possible to communicate functionality to users in a natural and unintrusive way [17].

Whenever people construct objects, they draw on a background of shared design language in their community and culture. Even something as apparently simple as a door is built to communicate to the user through convention. A door with a flat plate near shoulder level says, "push me!" One with a round knob says, "turn here", and one with a fixed graspable handle says, "pull." Although these messages are related to the underlying ability to perform the acts, they are also a matter of convention and learning, as every tourist finds out in trying to deal with everyday objects in an unfamiliar culture. We learn such languages from our everyday experience, and when a designer defies them, the result is confusion [15].

Design languages can be more or less natural, more or less intuitive (comprehensible to the user on the basis of previous expectations). As a simple example, a slider that moves horizontally can be used to control a dimmer on a light. If it were wired to make the light brighter when moved to the left and dimmer when moved to the right, it would confuse most users from a European culture. There are even some conceptual mappings and metaphors such as "up is more" that cut across all kinds of phenomena and often even across languages and cultures [9]. The designer of an artifact for interaction needs to harness these general cognitive resources and languages to the specifics of the particular interface at hand.

Just as a modern programming environment provides the programmer with a base language and libraries of common program elements, the design environment is populated by the collection of design languages on which the designer can draw in creating something new. This is in spite of all the lawsuits and concerns about look and feel infringement. The designer needs to be well versed in all of the common design languages and elements that users will encounter, either to employ them, or to avoid them if economic and legal concerns require that.

Genres/Styles. It is important to recognize that there isn't a best design language for interacting with computers, just as there isn't a best kind of building, or a best kind of literature. Every piece of software conveys a "genre," with its own language of expectations and interpretations. We are familiar with genres in literature (the Greek tragedy, the Victorian novel, the pulp romance), and architecture (the Greek temple, the Gothic cathedral, the post-modern office complex). The concept is equally applicable in computer software with the spreadsheet, the video game, and the word processor. The designer working within a background of experience in these genres can effectively use the expectations that go with them to situate the user in previous experiences and to move beyond them [3]. The power of genres is clear if we try to imagine a spreadsheet with a joystick and lifelike explosive sound effects whenever a formula is entered into a cell, or a word processor that requires the user to assemble words by chasing letters around on the screen.

Of course, the genre can never be taken as the boundary of what can be designed. Just as poets (and even technical writers) will creatively bend language to new purposes, the creative designer will mix, distort, and at times completely violate language conventions for a desired effect. KidPix, a drawing program, comes close to the purported counter-examples above, bringing video game design language elements (such as wacky sound effects and cartoon icons) into a drawing program. For the intended audience of young children, the mix is quite appealing. In effect, KidPix has introduced a new language which is now being duplicated in other products.

Participatory Design

Interactive debugging

Participatory design

All approaches to software engineering require a form of testing. Some methodologies call for carefully developed test suites and rigorous testing of components before and after integration into a larger program. As with the other issues addressed above, testing takes on a more complex and broader meaning for programs that do not just calculate a result from a few inputs, but which enter into a dynamic (and unpredictable) sequence of interactions with users. There are standard practices in the software industry for testing in use—alpha test, beta test, usability laboratories, and the like [13].

An environment for software design includes the tools for testing: both the technical tools (e.g., the observational technology of usability labs) and social tools (the people and practices required to identify, recruit and interact with testers at different levels and points in the design process). Going further, the process of interaction does not follow the classical "generate and test" where the designer develops a working program, then sends it off to users to test. The dialogue with users can begin with the first sketch of an idea and continue through all the stages in which the functionality as well as the interface is determined. The debugging starts with the ideas, not with the code. The environment for this dialog goes well beyond the workstations and files of the traditional programming environment [12, 21].

Expanding the Debugging Environment. There are limits to what can be learned about software while working in the programming office or the software testing laboratory. Some aspects, such as the speed and convenience of different interface mechanisms, can be tested to a high degree of accuracy. But others, often much more important, don't show up unless the user is in the natural context in which the system will be employed. What happens when the phone rings in the middle of an activity with the system? What if the person at the next desk is using a different word processor and you need to share a document? A designer who creates a system that works in idealized conditions may end up blaming (and alienating) the user when those ideal conditions don't exist in the chaotic realities of his or her life. A designer who can understand and anticipate the chaotic realities can produce a new level of usability.

To get people (both designers and users) to think about these interactions early in the design process, when they can most easily be taken into account, it is often important to interact in the actual setting where the final product will be situated. The insights for copier design that came from extensive field visits to see where and how the copiers were really used, and by whom, are described in [17]. Much of the success of the Quicken program for personal finance is attributed to an explicit "follow the user home" policy, in which the designers worked with people who purchased early copies to see what actually happened when they tried to install, use and integrate them into their everyday practices.

In the area of office software, this problem has been addressed through a method called "contextual inquiry," [6] in which the designer enters into the situated context of the user to learn about possibilities. An environment for the prototyping and conceptual design tools discussed needs to extend beyond the walls of the software organization to engage users in the process.

Use in Organizations. When we think of a piece of software on a personal computer, we tend to visualize the "user" as a person—an individual sitting in front of the machine. On the other hand, when we think of a traditional mainframe-based system, such as airline reservation system, inventory control system, or payroll system, there isn't a single prototypical individual user, but an organizational user composed of many people with different roles and functions. This distinction between personal and mainframe software is blurring in today's age of distributed client-server software, interconnected information networks, and groupware. The design of a computer application, regardless of its specific details, is intertwined with the design of the organizational interactions that surround its use.

Over the past few years, a number of approaches have emerged for looking at how computer system design interacts with organizational design and activities. Conferences, journals, and books have appeared on computer-supported cooperative work (CSCW), groupware, and organizational computing [5]. Of course, since the beginning of computing, people have used computers in group and organizational settings. The shift lies in asking the designer to focus on the way that design will affect people in those settings.

One interesting indicator is in the change of terminology used by information systems professionals. In earlier days they talked about the importance of "systems analysis"—getting a model of the organizational system before designing the information structures for it. Today we hear more in the management and information technology magazines about "business process re-engineering." The structures and practices of the business are not taken as a fixed environment to be analyzed and adjusted to, but as a domain of potential change and new design [22].

With this shift there has been increasing interest in what can be offered by those who bring lessons from systematic studies of the nature and structure of work to software and systems design. Anthropologists, ethnographers, sociologists, organizational theorists, and researchers in other disciplines have become a part of the interdisciplinary teams that approach the design of systems from a work-oriented perspective [4].

Many of the design activities outlined in the preceding sections have been advocated for over a decade *and yet are far from standard practice in the industry.*

Co-evolution of Practices, Tools, and Social Systems. The final extension of the dialogue with users is that the design cycle does not start and end with a product. The overall environment of computer use is a constant co-evolution in which new tools lead to new practices and ways of doing business, which in turn create problems and possibilities for technical innovation. We tend to think of the designers and programmers in the development laboratory as the primary part of the design environment, but in this larger picture, the people in customer support are also central participants in the dialog. Many companies are beginning to integrate this critical source of feedback into the design cycle explicitly; some companies even require that system designers spend a significant amount of time in a help-desk role, to see the consequences of their designs in actual use.

The Designers' Organizational Environments. Many of the design activities outlined in the preceding sections have been advocated for over a decade and yet are far from standard practice in the industry. The picture is oversimplified in its implicit notion of a designer or design team, working in concert to produce software. In fact, the software organization contains many disjointed parts concerned with software design, from marketing to interface design and development, to customer support, training, and documentation. The coordination of these often far-flung groups with diverse interests and responsibilities makes it a very different matter to put into real practice the theoretical practices that go into our design environment [16]. In many cases, the most significant elements in creating a productive environment for software design are the organizational structures and changes that need to be made in order to support the communication and flow of activities that constitutes software design.

Conclusion

The environment for the designer goes well beyond the traditional bounds of programming environments. In fact the preceding descriptions may feel to many readers like a boundless extension—opening up the concerns of the designer to so many issues and methods that nothing can ever get done. Of course, not every environment or every piece of software requires explicit attention to all the dimensions of design. A project to port an email interface from one window system to another may require careful attention to design languages, but it can take for granted most of the initial analysis of the setting and patterns of use. An attempt to create a totally novel kind of application may require situated observation of what people do and how their lives might be changed by a new technology, but it may be the kind of application that doesn't require careful study of how the new software will modify practices of a group.

Taking a broad view will prompt awareness—awareness on the part of software designers of the issues they need to think about, and awareness on the part of those who create environments (computational, physical, and social) for those designers of the objects and methods they need to support. As with all tools, there is no magic—the environment does not produce the result. But a comprehensive and thoughtfully constructed environment can facilitate the human creativity that is always at the core of design.

References

1. Adler, P. and Winograd, T., Eds. *Usability: Turning Technologies into Tools.* Oxford, 1992.
2. Apple Computer. *Human Interface Guidelines: The Apple Desktop Interface.* Addison-Wesley, Reading, Mass., 1987
3. Brown, J. and Duguid, P. Borderline issues: Social and material aspects of design. *Human-Computer Interaction 9,* 1 (1994), 3–36.
4. Greenbaum, J., and Kyng, M. *Design at Work: Cooperative Design of Computer Systems.* Erlbaum, NJ, 1991.
5. Grudin, J., Ed. Special issue on Collaborative Computing. *Commun. ACM 34,* 12 (Dec. 1991), 30–34.
6. Holtzblatt, K., and Jones, M. Contextual inquiry: A participatory technique for system design. In D. Schuler and A. Namioka, Eds., *Participatory Design: Principles and Practices.* Erlbaum., NJ, 1993, 177–210.
7. Johnson, J., Roberts, T., Verplank, W., Smith, D. C., Irby, C., Beard, M., and Mackey, K. Xerox Star, a retrospective.z *IEEE Comput.* (Sept. 1989), 11–26.
8. Kapor, M. A software design manifesto: Time for a change. *Dr. Dobb's Journal* 172 (January 1991), 62–68.
9. Lakoff, G., and Johnson, M. *Metaphors We Live By.* University of Chicago Press, Chicago, 1980.
10. Laurel, B. *The Art of Human-Computer Interaction.* Addison-Wesley, Reading, Mass., 1990.
11. Moran, T. Introduction to the special issue on Context in Design. *Human-Computer Interaction 9,* 1 (1994), 1–2.
12. Muller, M., and Kuhn, S., Eds. Special issue on participatory design. *Commun. ACM 36,* 6 (June 1993), 24–28.
13. Nielsen, J. *Usability Engineering.* Academic Press, 1993.
14. Norman, D. Cognitive engineering. In D. Norman and S. Draper, Eds., *User Centered System Design: New Perspectives on Human-Computer Interaction.* Erlbaum, NJ, 1986, pp. 31–62.
15. Norman, D. *The Design of Everyday Things.* Basic Books, 1988.
16. Poltrock, S., and Grudin J., Organizational obstacles to interface design and development: Two participant-observer studies. *ACM Trans. Computer-Human Interaction 1,* 1 (Mar. 1994), 52–80.
17. Rheinfrank, J., Hartman, W., and Wasserman, A. Design for Usability: Crafting a strategy for the design of a new generation of Xerox copiers. In P. Adler and T. Winograd, Eds., *Usability: Turning Technologies into Tools.* Oxford, 1992, 15–40.
18. Rheinfrank, J., and Hefley, A. Reflections. *Interactions 1,* 1 (Jan. 1994), 88.
19. Schön, D. *The Reflective Practitioner.* Basic Books, 1983.
20. Schrage, M. The Culture(s) of Prototyping. *Design Management J. 4,* 1 (Winter 1993), 55–56.
21. Schuler, D. and Namioka, A. Eds., *Participatory Design: Principles and Practices.* Erlbaum, NJ, 1993.
22. Winograd, T. and Flores, F. *Understanding Computers and Cognition: A New Foundation for Design.* Addison-Wesley, Reading, Mass., 1987.
23. Winograd, T., Ed. *Designing Software Interactions.* Addison-Wesley, in press.

About the Author:
TERRY WINOGRAD is a professor of computer science at Stanford University. Current research interests include human-computer interaction, software design, and digital libraries. **Author's Present Address:** Stanford University Department of Computer Science, Stanford, CA 94305-2140; email: winograd@cs.stanford.edu

If *you would like to pursue cognition and software development in more depth, the proceedings from one or more of the five workshops on Empirical Studies of Programmers (published as the Ablex Empirical Studies of Programmers Series, Norwood, NJ) are good sources of information. The sixth workshop will be held in Washington, DC in January, 1996. ACM has several special interest groups (SIGs) pertaining to this topic, with computer-human interaction (SIGCHI) the broadest in scope. You might also join the international Psychology of Programming Interest Group (PPIG), an informal group that has a mailing list and a newsletter. Its seventh annual workshop will be in Edinburgh, also in January, 1996. Join the PPIG mailing list by sending a request to ppig-request@ee.surrey.ac.uk, and read the PPIG Web page at http://www.u-net.com/ppig/.*