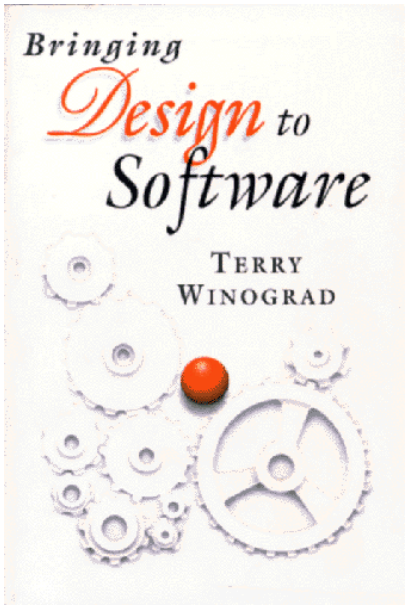

Bringing Design to Software

1

Mitchell Kapor

A Software Design Manifesto

The most important social evolution within the computing professions would be to create a role for the software designer as a champion of the user experience.... What is design? ... It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together.

Mitchell Kapor was one of the first people in the microcomputer industry to identify his work as *designing software*. When he began work on Lotus 1-2-3 in the early 1980s, he took on the task of designing the interactions, but he wasn't the programmer. He worked closely with a skilled programmer, Jonathan Sachs, interactively developing the design at all levels of detail—from the broad architecture to the structured organization of the command menus and the naming of the menu items.

He recounts the story of being asked by his son what he did for a living, and of being unable to come up with a good answer—he wasn't a programmer, but he was developing programs. On the other hand, he wasn't a manager, in that he was doing the detailed design work himself, rather than directing other workers. He was doing software design, but he didn't have a label for that kind of work. In his manifesto, which is reproduced here, and in a number of other talks and writings over the past few years, he has eloquently made the case that we need to think of software design as a *profession*, rather than as a side task of a manager or a programmer.

Kapor delivered his manifesto in 1990 at Esther Dyson's PC forum, a renowned gathering of microcomputer industry leaders. The response ranged from strong enthusiasm to the predictable "Why are you complaining—we're selling lots of software?" It first appeared in print 1 year later (Kapor, 1991) in *Dr. Dobbs Journal,* one of the oldest and most widely read magazines for microcomputer programmers.

This chapter is the only one in the book that is reprinted from an earlier publication. As a call to arms, it has in important place in helping us to understand the history and context of our field. The points that Kapor made are still as valid today as they were a few years ago, and the themes that he introduced echo throughout this book.

— Terry Winograd

§ § §

The great and rapid success of the personal computer industry over the past decade is not without its unexpected ironies. What began as a revolution of individual empowerment has ended with the personal computer industry not only joining the computing mainstream, but in fact defining it. 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, which means that the revolution, measured by its original goals, has not as yet succeeded.

Instead we find ourselves in a period of retrenchment and consolidation in which corporations seek to rationalize their computing investment by standardizing on platforms, applications, and methods of connectivity, rather than striving for a fundamental simplification of the user experience. In fact, the need for extensive help in the installation, configuration, and routine maintenance of system functions continues to make the work of corporate data processing and MIS departments highly meaningful. But no one is speaking for the poor user.

There is a conspiracy of silence on this issue. It's not splashed all over the front pages of the industry trade press, but we all know it's true. Users are largely silent about this. There is no uproar, no outrage. Scratch the surface and you'll find that people are embarrassed to say they find these devices hard to use. They think the fault is their own. So users learn a bare minimum to get by. They underuse the products we work so hard to make and so don't help themselves or us as much as we would like. They're afraid to try anything else. In sum, everyone I know (including me) feels the urge to throw that infuriating machine through the window at least once a week. (And now, thanks to recent advances in miniaturization, this is now possible.)

The lack of usability of software and the poor design of programs are the secret shame of the industry. Given a choice, no one would want it to be this way. What is to be done? Computing professionals themselves should take responsibility for creating a positive user experience. Perhaps the most important conceptual move to be taken is to recognize the critical role of design, as a counterpart to programming, in the creation of computer artifacts. And the most important social evolution within the computing professions would be to create a role for the software designer as a champion of the user experience.

By training and inclination, people who develop programs haven't been oriented to design issues. This is not to fault the vital work of programmers. It is simply to say that the perspective and skills that are critical to good design are typically absent from the development process, or, if present, exist only in an underground fashion. We need to take a fresh look at the entire process of creating software—what I call the *software design viewpoint*. We need to rethink the fundamentals of how software is made.

The Case for Design

What is design? What makes something a design problem? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. Consider an example.

Architects, not construction engineers, are the professionals who have overall responsibility for creating buildings. Architecture and engineering are, as disciplines, peers to each other, but in the actual process of designing and implementing the building, the engineers take direction from the architects. The engineers play a vital and crucial role in the process, but they take their essential direction from the design of the building as established by the architect.

When you go to design a house you talk to an architect first, not an engineer. Why is this? Because the criteria for what makes a good building fall substantially outside the domain of what engineering deals with. You want the bedrooms where it will be quiet so people can sleep, and you want the dining room to be near the kitchen. The fact that the kitchen and dining room should be proximate to each other emerges from knowing first, that the purpose of the kitchen is to prepare food and the dining room to consume it, and second, that rooms with related purposes ought to be closely related in space. This is not a fact, nor a technical item of knowledge, but a piece of design wisdom.

Similarly, in computer programs, the selection of the various components and elements of the application must be driven by an appreciation of the overall conditions of use and user needs through a process of intelligent and conscious design. How is this to be done? By software designers.

Design disciplines are concerned with making artifacts for human use. Architects work in the medium of buildings, graphic designers work in paper and other print media, industrial designers on mass-produced manufactured goods, and software designers on software. The software designer should be the person with overall responsibility for the conception and realization of the program.

The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight.

The same might be said of good software. Firmness: A program should not have any bugs that inhibit its function. Commodity: A program should be suitable for the purposes for which it was intended. Delight: The experience of using the program should be pleasurable one. Here we have the beginnings of a theory of design for software.

Software Design Today

Today, the software designer leads a guerrilla existence, formally unrecognized and often unappreciated. There's no spot on the corporate organization chart or career ladder for such an individual. Yet time after time I've found people in software development companies who recognize themselves as software designers, even though their employers and colleagues don't yet accord them the professional recognition they seek.

Design is widely regarded by computer scientists as being a proper subpart of computer science itself. Also, engineers would claim design for their own. I would claim that software design needs to be recognized as a profession in its own right, a disciplinary peer to computer science and software engineering, a first-class member of the family of computing disciplines.

One of the main reasons most computer software is so abysmal is that it's not designed at all, but merely engineered. Another reason is that implementors often place more emphasis on a program's internal construction than on its external design, despite the fact that as much as 75 percent of the code in a modern program deals with the interface to the user.

More Than Interface Design

Software design is not the same as user interface design.

The overall design of a program is to be clearly distinguished from the design of its user interface. If a user interface is designed after the fact, that is like designing an automobile's dashboard after the engine, chassis, and all other components and functions are specified. The separation of the user interface from the overall design process fundamentally disenfranchises designers at the expense of programmers and relegates them to the status of second-class citizens.

The software designer is concerned primarily with the overall conception of the product. Dan Bricklin's invention of the electronic spreadsheet is one of the crowning achievements of software design. It is the metaphor of the spreadsheet itself, its tableau of rows and columns with their precisely interrelated labels, numbers, and formulas—rather than the user interface of VisiCalc—for which he will be remembered. The look and feel of a product is but one part of its design.

Training Designers

If software design is to be a profession in its own right, then there must be professional training that develops the consciousness and skills central to the profession.

Training in software design is distinguished from computer science, software engineering, and computer programming, in that its principal focus is on the training of professional practitioners whose work it is to create usable computer-based artifacts—that is, software programs. The emphasis on developing this specific professional competency distinguishes software design on the one hand from computer science, which seeks to train scientists in a theoretical discipline, and on the other, from engineering, which focuses almost exclusively on the construction of the internals of computer programs and which, from the design point of view, gives short shrift to consideration of use and users.

In architecture, the study of design begins with the fundamental principles and techniques of architectural representation and composition, which include freehand drawing, constructed drawing, presentation graphics, and visual composition and analysis.

In both architecture and software design it is necessary to provide the professional practitioner with a way to model the final result with far less effort than is required to build the final product. In each case specialized tools and techniques are used. In software design, unfortunately, design tools aren't sufficiently developed to be maximally useful.

Hypercard, for instance, allows the ready simulation of the appearance of a program, but is not effective at modeling the behavior of real-world programs. It captures the surface, but not the semantics. For this, object-oriented approaches will do better, especially when there are plug-in libraries, or components, readily available that perform basic back-end functions. These might not have the performance or capacity of back ends embedded in commercial products, but will be more than adequate for prototyping purposes.

A Firm Grounding in Technology

Many people who think of themselves as working on the design of software simply lack the technical grounding to be an effective participant in the overall process. Naturally, programmers quickly lose respect for people who fail to understand fundamental technical issues. The answer to this is not to exclude designers from the process, but to make sure that they have a sound mastery of technical fundamentals, so that genuine communication with programmers is possible.

Technology courses for the student designer should deal with the principles and methods of computer program construction. Topics would include computer systems architecture, microprocessor architectures, operating systems, network communications, data structures and algorithms, databases, distributed computing, programming environments, and object-oriented development methodologies.

Designers must have a solid working knowledge of at least one modern programming language (C or Pascal) in addition to exposure to a wide variety of languages and tools, including Forth and Lisp.

The Software Design Studio

Most important, students learn software design by practicing it. A major component of the professional training, therefore, would consist of design studios in which students carry out directed projects to design parts of actual programs, whole programs, and groups of programs using the tools and techniques of their trade.

Prospective software designers must also master the existing research in the field of human–computer interaction and social science research on the use of the computer in the workplace and in organizations.

A design is realized only in a particular medium. What are the characteristic properties of the medium in which we create software?

Digital media have unique properties that distinguish them from print-based and electronic predecessors. Software designers need to make a systematic study and comparison of different media—print, audiovisual, and digital—examining their properties and affordances with a critical eye to how these properties shape and constrain the artifacts realized in them.

Design and the Development Process

Designers must study how to integrate software design into the overall software development process—in actual field conditions of teams of programmers, systems architects, and technical management.

In general, the programming and design activities of a project must be closely interrelated. During the course of implementing a design, new information will arise, which many times will change the original design. If design and implementation are in watertight compartments, it can be recipe for disaster because the natural process of refinement and change is prevented.

The fact that design and implementation are closely related does not mean that they are identical—even if the two tasks are sometimes performed by one and the same person. The technical demands of writing the code are often so strenuous that the programmer can lose perspective on the larger issues affecting the design of the product.

Before you can integrate programming and design, each of the two has to have its own genuine identity.

A Call to Action

We need to create a professional discipline of software design. We need our own community. Today you can't get a degree in software design, go to conference on the subject, or subscribe to a journal on the topic. Designers need to be brought onto development teams as peers to programmers. The entire PC community needs to become sensitized to issues of design.

Software designers should be trained more like architects than like computer scientists. Software designers should be technically very well grounded without being measured by their ability to write production-quality code.

In the year since I first sounded this call to action, there has been a gratifying response from the computing industry and academic computer science departments. At Stanford University, Computer Science Professor Terry Winograd has been awarded a major National Science Foundation grant to develop and teach the first multicourse curriculum in software design. And in Silicon Valley and elsewhere there is talk of forming a professional organization dedicated to advancing the interests of software design.

Suggested Readings

Nathaniel Borenstein, *Programming as if People Mattered: Friendly Programs, Software Engineering, and Other Noble Delusions,* Princeton, NJ: Princeton University Press, 1991.

Paul Heckel, *Elements of Friendly Software Design,* Berkeley, CA: Sybex, 1994.

Bruce Tognazzini, *Tog on Software Design,* Reading MA: Addison-Wesley, 1995.

About the Author

Mitchell Kapor was the founder of Lotus Development Corporation and the designer of the Lotus 1-2-3 spreadsheet. Presently, he is an adjunct professor in the Media Laboratory at MIT, developing courses on software design, and is chair of the advisory board of the Association for Software Design. He is also a cofounder and board member of the Electronic Frontier Foundation.