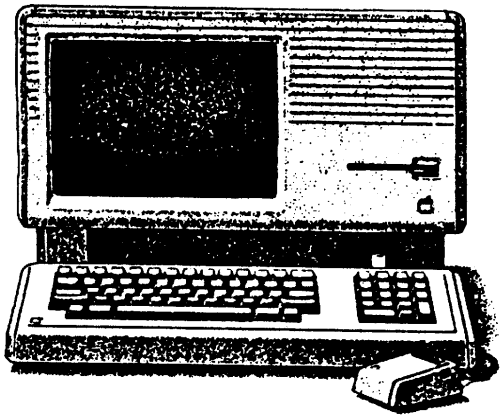
Doc # 136

Apple Lisa Information



FILE NAME

Enlisting User Help in Software Design

DISK #

COMMENTS

Apple/Larry Tesler

David T. Craig
736 Edgewater, Wichita, Kansas 67230
(316) 733-0914

The Lisa™ Professional

"136-00.PICT" 261 KB 2000-12-24 dpi: 400h x 400v pix: 2555h x 3697v

January 1983 SIGCHI BULLETIN Volume 14,3

ENLISTING USER HELP IN SOFTWARE DESIGN

Pages: 5

Larry Tesler
Personal Office Systems Division
Apple Computer, Inc.

1. HISTORY

I first got into the area of user-friendly software around 1963, when I helped to develop a graphics language for art students. The job of the art students was to design card stunts for half-time shows at college football games. Our graphics language allowed them to encode their design in numeric terms. We fed their descriptions to a computer program which then prepared an instruction card for each seat in the rooting section.

During the project, I learned to appreciate that right-brain artists think differently from left-brain computer scientists. It was necessary to understand their thought processes in order to design a usable graphics language.

The next time that I got really involved in this area was at Xerox PARC from 1973 to 1975. We were trying to develop an office automation system based on personal workstations such as the Xerox Alto computer. We were very excited about the possibilities of large displays with a mouse as a pointing device. Our initial user interface ideas were adopted from the NLS system developed at SRI International. (NLS later evolved into the Augment system offered by Tymshare, Inc.)

Despite the enthusiasm of the developers, there was a lot of resistance among office workers who tried our system. To find out why, I developed a way to watch people at work, observe what difficulties they were having, and elicit explanations from them.

One observation made during those tests later proved to be a key idea. A lay person trying a system for the first time is probably feeling pretty stupid and thinks that any mistake is his or her own. His inclination is not to tell you what is difficult because he thinks it is his problem. It is important to keep emphasizing that if he is having a problem, it is not his but the computer's. If you get into a car, turn the wheel, and it steers about twice as far as you expect, it is not that you are an incompetent driver, it's because the car has oversteer.

2. WHEN TO ENLIST USER HELP

Let me talk a little about my work at Xerox PARC from 1973-1980 and at Apple since 1980, where I have been designing interactive systems. I brought the user into the design process as a helper, and the results were extremely good.

There are two stages of the development process during which I think it is critical to get user help.

One critical time is during the design phase, when the designer has an idea of what his goals are and probably has formulated some idea of what the user interface of his application is going to be. But before the design is completely frozen, it is a good idea to get a reality check. At that point, my design teams go out and talk to potential users; we call it "interviewing at the workplace."

The other critical time is after the software has been implemented to the prototype stage, before it is frozen, when there are still some opportunities to make changes. Since I am unable to produce armchair software designs that are really usable, I have found it is really important to build and verify prototypes, and make numerous changes. The technique is very similar to what Tom was telling you about, "formative evaluation." *

3. ENLISTING USER HELP DURING THE DESIGN

During the formative stages of the development effort, the designers need to get user input. If they wait until the product is ready to ship, it is too late. In the projects I have worked on, we go to the workplaces of potential users during the early stages of the design to observe how people actually work.

3.1 Interviewing at the Workplace

Up to three people from the design team observe and interview a prospective user. The goal of each interview is to elicit from the prospect his or her:

- Procedures
- Problems
- Forms and Tools
- Terminology

* Editor's note: see page 13.

Even if we are just designing a text editor, we want to know how the documents the prospect produces fit into the entire process of his job. Where does the paperwork he is typing come from, and where does it go to after it leaves his desk? How many revision cycles are there, who specifies the revisions, and what does marked-up copy look like?

If we are designing a data base application, we want to know how he keeps things in his office. Does he use index cards, files, or what?

If the prospect handles nine-part purchase order forms, what happens to each of the nine parts, who fills them in, and where are they routed?

It turns out that by asking very open-ended questions we can find out things that we did not even anticipate--usually very surprising things. We designers tend to be limited by our own experience and by the conventions of the data processing industry, which really do not reflect the way other people work.

It is also important to find out what terminology people use. For example, in the Apple UCSD Pascal Text Editor, there is a command called "Save on File." Our secretaries seemed to have no difficulties using this command, so people theorized that it was perfectly fine to call a document you are preparing with a text editor a "file." But we also noticed that this was not really the way people in offices talked about things. They did not call a document a "file;" a file was a folder or a cabinet.

We were puzzled about why people did not have any trouble with the term "file" in the editor, so we asked several secretaries with experience using the editor what the word "file" meant on the computer. All of them said a file was a diskette. We then asked them the meaning of the term "file" in the command "Save on File Named X." Their interpretation of this phrase was "Save my memo on the the file [the diskette] and name it [my memo] X." The fact that the wording of the command contradicted their notion did not change their idea of what a file was; instead, they changed their parsing of the command so it would make sense.

One more point. During a user interview, we focus on current practice. We don't ask, "If you had a computer, what would you do with it?" Once in a while, this will elicit an interesting remark. But people are not very good at visualizing what it would be like to have a computer in front of them, so this is not a very productive approach.

4. ENLISTING USER HELP DURING IMPLEMENTATION

4.1 Prototype Testing

After developing prototype software, we return to the users again to get their help in verifying that the prototypes have good human factors. I would say that in all cases I have ever tested, they do not! We discover what problems there are and allow time to correct them.

In a one year development, if we have a fairly good first cut at a user interface, maybe two or three months need to be left at the end to fix most of the problems that may be found. We improve the human factors at the same time that we are fixing bugs, improving performance, and implementing extra features.

We really do not need to have a final prototype in order to do the test. If we have something that demonstrates the major functions and if a user can get through a session without likelihood of a crash, it is good enough for our purposes.

The goals of prototype testing are:

- To resolve specific design issues;
- To uncover unanticipated problems;
- To find out what is easy and what is hard to learn, and what is smooth and what is clumsy to use.

4.2 Resolving Specific Design Issues

Let me describe an example of a design issue that has recurred several times in my projects.

Some of you may have read my Byte magazine article (August 1981) about how to eliminate modes from systems. Findings from my early studies at Xerox PARC showed that modes were very difficult for people to deal with. People were always getting into the wrong mode and did not know how to get back out.

My colleagues and I found an easy way to eliminate modes from most commands in a text editor. We structure the command language so that the user first selects the operand of an operation and then issues the command to operate on it. Unfortunately, if one wants to move or copy something from one place to another, it requires two operands, the source and the destination. The simple technique only works when there is one operand.

Inevitably, someone would propose that we allow two selections to exist at a time. The user would select both a source and a destination,

using different buttons on the mouse, if there is a mouse. There would be no mode at any point because he could continually respecify each selection independently. When both selections were finally to his liking, he could issue the Move command or the Copy command.

This idea is a very appealing one. One colleague dubbed it the "Place and Thing Model." The Thing is what you want to move or copy and the Place is where you want it to go. An obvious corollary of the model is that typing is inserted at the Place while the Delete command deletes the Thing.

Fortunately, we tested the Place and Thing Model before we shipped it to anybody. We implemented a prototype and ran user tests. Actually, I had to repeat the experiment several times over the years for the benefit of people who did not believe the results.

We were shocked (the first time) to find that people were confused by the Place and Thing interface. It became clear after talking with them and also by watching the errors they made that the confusion arose from having two foci of attention. They couldn't predict whether something they typed would go to the Place or to the Thing, nor whether the Underline command would underline the Thing or would underline new text typed at the Place. In other words, the Place and Thing Model made copying and moving easier, but it made simple operations like typing and underlining more difficult.

To solve the move-and-copy problem, the Star design team at Xerox instituted a temporary mode while the user was waiting to specify a destination, and the Smalltalk design team instituted a two-step cut and paste scheme. Both proved less error-prone than the Place and Thing paradigm.

The Place and Thing experience demonstrates that even an idea that seems elegant and that satisfies some principle is not necessarily going to succeed. The only way to find out is to test it.

4.3 Uncovering Unanticipated Problems

Besides resolving specific design issues, prototype testing always uncovers unanticipated problems. For every design issue we knew about, we would find a dozen unanticipated problems with the system. People made too many errors, did not understand what they had done, and were confused by the terminology.

I think uncovering unanticipated problems is the greatest benefit of prototype tests. When we focused too much on known issues, the tests rarely were worth the time and trouble they took.

4.4 Qualifying Subjects

The people we like to observe in our prototype tests are novice users who are prospective end-users of the system. For example, if we are testing out accounting software, we try to find accounting clerks that have very little experience.

It is becoming difficult these days to find people who have never used a computer. In 1973, we simply brought in secretaries from temporary agencies. At Apple, we ask new employees at orientation to fill out a form summarizing their background and noting any previous use of computers.

It is important to know the person's experience. Have they used video games? Have they used data-entry systems? Do they have a personal computer at home, or have they taken any computer classes? The results of the test will be interpreted differently depending on their experience. For example, if we find someone constantly hitting the return key when it is not necessary, he may have once used a system that required it.

4.5 Beginning a Prototype Test

We schedule an hour for each test. At the beginning of the test, it is important to get the person relaxed and comfortable and to emphasize that it is not we who are testing him, it is he who is testing the machine. We say, "The machine is new and is just a prototype, so it has lots of bugs in it. It is not very easy to use yet, and we would like you to help us figure out how to make it easier." This assurance helps to alleviate the feeling that he is being tested, and that he is at fault when he makes mistakes.

Before beginning, we try to create a positive mind set before the user encounters difficulties. The general approach is to assure him that the system is not going to be hard, although he will have an occasional problem. However, there are aspects of most systems that are very difficult for all users. If we told a user that those aspects were going to be easy, and then he made the inevitable mistakes, he would feel it was his own shortcoming. In such cases, I take a different tactic. I tell the user that this aspect of the system is going to be very hard to learn, but by the end of the hour, he will probably master it. Then after fifteen or twenty minutes, when he begins to get a handle on it, he feels accomplished and positive about his ability to master the system.

4.6 Conducting a Prototype Test

During the test, I sit down next to the user and tutor him step by step in how to use the software. I try never to touch the machine myself. Being present, I can vary the instruction as I go along in response to what he is doing. Even though I plan a certain scenario, something may occur during the test that suggests a more fruitful experiment. I will change the scenario and explore the problem more deeply.

During the test, I note every event, whether a success or a failure. If I only noted errors, the subject would soon view my notetaking as an intimidating act.

I try to talk as little as possible during the test. I give a brief introduction and get the user working immediately so he does not feel as though he is being lectured. Sometimes, I give more detailed instructions, but if I talked him through every step he would just mechanically execute my instructions and not recall much. Instead, I let him practice and play with what he has learned.

The user should not only execute tasks correctly, he should also understand why they are done the way they are so he can later perform them without assistance. Frequently, I stop giving instructions and let him guess and try things, rather than telling him every step. But during the first few minutes, it is helpful to actually drive him through each step to help him overcome any initial hesitation.

4.7 Maximizing Test Benefits

To make the tests of maximum benefit to the designers, there are certain rules we try to follow:

- Don't offer help too soon;
- Welcome suggestions and reactions;
- Observe but don't pressure.

If the person seems stumped for a moment, I let him think a little while to see if he can figure out for himself what to do. If he starts getting to the point where he is feeling uncomfortable, then I will finally intervene.

It is important to be open to the user's suggestions and reactions. If the person running the test gets defensive, argues, or tries to put off discussion until later, the user will clam up right away. Even if the subject makes an obvious statement, a trivial remark, or suggests something that is impossible to do on a computer, it is important to welcome his reactions and discuss them as an equal. Otherwise, he will stop telling you the things you need to hear.

4.8 Interviewer Qualifications

Running prototype tests is not easy. I have been doing it for nine years and I still learn from my mistakes. When I conduct a series of tests on the same software, I get better at teaching and observing subjects with each subsequent test.

During an interview, it is important to be objective, observant, sensitive, and patient. Someone who wants to run a prototype test must be willing to assume that attitude, as well as be a competent teacher.

One of the less obvious requirements for running a prototype test is familiarity with the software. We often have psychologists run user tests. Ours have the right attitude, are good teachers, and have previous experience running subjects in experiments. But when they tried to test early prototypes, they found themselves insufficiently familiar with the software. If the user made an error, the observing psychologist did not understand the error or did not even detect that an error had been made. If the user wanted to explore something outside the scenario, the teacher was stuck. So for early prototype testing, it is important to have someone present who is familiar with the software.

When one individual does not have all of the required skills, we run tests with two or three people in the room: a teacher who knows the software, an observer who knows it even better, and perhaps a psychologist who can observe from a different perspective and intervene if the others become impatient or insensitive.

4.9 Concluding a Prototype Test

At the end of each test, we take time to gather impressions from the subject. We start out asking open questions. "Well, what do you think?" or "What was hard; what was easy?" Later we pose more direct questions like "The time you made that certain error, what were you thinking?" or "What do you think is a better term for this concept?" Usually in the open-question stage we get some really good responses that are unexpected. The directed questions address more specific issues.

If there is time after the question period, I ask the subject to go back and do some more tasks. He or she has been using the machine only an hour, so the question period is needed to "cleanse" his short-term memory. After that, having him try to repeat certain tasks tests his recall of what he has been taught. This

provides us with clues about what is easy and what is hard to remember.

4.10 Homework Between Tests

Between tests, the teacher has homework to do. If there were other people observing the test, we compare notes and debate interpretations. I reevaluate the teaching approach and plan the issues and the scenarios for the next test. If the first test takes an hour, usually I can cover the same material in a half hour the second time and thus have additional time to teach more features.

After I have run through a whole series of tests on an application, I write a report about what was commonly experienced by the users. Generally, most users experience the same things. Occasionally, some user's unique quirk may be worth mentioning, but what is of most concern is what most or all users experienced.

4.11 Employing Test Results

After an application is fully tested, we disseminate the results. Within Apple's organization everyone gets into the user interface act: marketing, training, technical writing, product design, product support, quality assurance, software, and management. Everyone needs to know the test conclusions to understand why changes are or are not being made. Then, of course, you must change the software as needed or else it was all for naught.

Our training group, staffed by psychologists and educators, runs a similar series of tests in order to evaluate training materials. Usually, we have already tested out the worst aspects, and by the time they get the software from engineering, they can debug their training materials using the same technique.

5. CONCLUSIONS

In conclusion, observing users really does work. Not only does it work, but it is really necessary. In my experience, only occasionally have I found software that has been really easy to use. In just about every case in which I was able to question the designers of friendly software, they said that they, too, ran extensive user tests and made extensive revisions before the software was released.

Some people think they can design friendly software just by being clever or by reading papers on principles of user interface design. Principles are important to consider when you are first designing, but unless you take the car out for a road test, you won't know about the oversteer.

FINIS