

Testing: “What, Me Worry?”

— written for *The X Journal*, 1996



Eric Schaffer, Ph.D., CUA, CPE, is Founder and CEO of Human Factors International, Inc. (HFI). He teaches, consults, and speaks on corporate and governmental Web and GUI interface design issues.



John Sorflaten, Ph.D., CUA, CPE, teaches and consults as a Project Director at HFI. With Eric, he initiated a usability curriculum at a local university in his home town of Fairfield, IA.

©1996, Human Factors International, Inc.

The insouciant, Cheshire-cat smile of Alfred E. Newman characterizes the penetrating satire of *Mad Magazine's* writers and editors. You recall bad boy Alfred's by-line: “What, Me Worry?” Now *Mad Magazine* appears on TV. Too bad we don't have time to watch it. We're too busy fixing applications developed with Alfred's philosophy. “What, Me Worry?” substitutes for adequate usability testing in the development life-cycle of too many corporate applications.

Test Early Good screen design requires usability testing early in the development cycle. Designers try substitutes that all fail. For example, dependence on user comments fails. Dependence solely on user representatives and solely on design walkthroughs fail. Usability outcomes from beta testing fail too, because beta results are too late. Usability changes become too costly to implement.

Usability testing early in the design phase can save you so much time, money, grief, and unneeded conversations (arguments), that the current failures of corporate America deserve as much satire as we can invent.

“What, Me Worry?” characterizes an ostrich syndrome: “What I don't see can't hurt me.” Many managers in a hurry invoke ostrich mentality to solve their schedule problems. In our design world, we dub it “cryptotesting.” Recall that cryptodesign manifests when a developer uses a design solution suitable for one problem, but misapplies it to another, totally different situation. Instead, use soul design to get it right.

Test Right Cryptotesting says, in effect, “as long as I feel good, then the application is good.” This kind of testing is great for dating or marriage(!), but a hazard for application development. This attitude shows us that the developer has sold his or her soul to the “feel good test.” (See Figure 1.) Neurologists (brain doctors) have a name for this disregard of objective evidence: anosognosia, meaning “disease of knowledge.” Let's see how it manifested in one medical case from the US Supreme Court.

THE ANOSOGNOSIA OF JUSTICE DOUGLAS In 1975, Supreme Court Justice William O. Douglas suffered a stroke that affected the right hemisphere of his brain. Since language functions reside in the left hemisphere, Justice Douglas continued to speak as fluently as ever. Therefore, even though the left side of his body was paralyzed from the stroke, everyone expected this brilliant and decisive member of the Court to continue his work on the bench. However, events slowly gave

Only 18% of riders could find the time of departure of a bus on the published schedule.

Survey of 2000 adults in Oregon

Testing an application for printing tabs: only one subject was able to complete it properly. That subject was a rocket scientist.

—HFI test protocol, 1995



Figure 1. Here's two instances in which users became test subjects after product release. Clearly, "What, Me Worry?" attitudes governed the development process. Software for printing out notebook tabs suffered competitively. We found out why with a simple test. Remember, it costs 60-100 times as much to make changes after product release compared to changes during project definition. Early testing pays off.

contrary evidence. First, he checked himself out of the hospital against medical advice (he did this several times). Then he trivialized his physical limitations, attributing his hospitalization to “a fall” and discounting his paralysis as “a myth.” After being confronted with the fact he could not walk or get out of his wheelchair, he said “Walking has very little to do with the work of the Court.” Still chair-bound, he later invited reporters to go hiking with him the following month. He repeatedly failed to follow social convention with other justices and staff. Unable to do his job, he refused to resign. And after being forced to resign, he often acted as if he still had the job.

The “Disease of Knowledge” Justice Douglas, it turned out, suffered defects in reasoning and emotions from his stroke, although his speech appeared normal. Luckily, other people could easily see that he denied his paralysis, thus bringing questions about his other decisions. Imagine if there were no paralysis for him to deny. He might have appeared

entirely normal, yet still remain afflicted with anosognosia—the “disease of knowledge.” Diagnosis would be doubly difficult. (Such cases occur.) Alfred E. Newman must be one of those patients, looking normal, but blissfully lacking insight into any risk he incurs. It won't be easy to tell Al the truth. This sounds ominously like cryptodesign at its deadliest. We call it “cryptotestiness”—a grouchy refusal to spend money to test usability early in the life cycle—a false economy.

CRYPTOTESTING SNEAK ATTACK Even worse than cryptotestiness, we have a sneak attack from the “junk food” of testing—user preferences. This empty calories approach to testing consists of asking users what they think about a screen. Managers think they get the benefits of testing. But they fail to realize that users simply don't have knowledge about important detailed design issues such as we covered previously in this column (see previous articles).

User Preferences Ergonomics researchers have studied the dichotomy of user preference versus user performance, often finding that users don't know what provides the best performance. For example, Dr. Robert Bailey, our Chief Scientist at Human Factors International, published results indicating users failed to discriminate between three levels of consistency in interface design. They had no preference. However, they performed better on the most consistent interface. Therefore, had the designer adopted user preference to guide the design (junk testing), the outcome could easily have been an inferior interface.

Even worse, clearly stated user preference can contradict performance results. Researchers Mack and Lang found that users preferred using a stylus and mouse for precision pointing. However, these methods created far more errors than performance with a keyboard-command input method. Togna-

Never Ask a User How They Would Design It!

If they knew, they would have your job.



Fig 2. User preference often fails to create detailed designs with the best performance. Design requires specialized knowledge. However, users can give you feedback on how well your design supported their workflow and functional requirements, areas in which they have a lot of experience.

zinni conducted a study in which subjects thought that using cursor keys was faster than a mouse to replace text in a word processing task. However, subjects using a mouse performed twice as fast as the cursor key condition! In both cases, designs adopting user preference would have resulted in a far less efficient product (see Figure 2).

Soul Testing In cryptotesting, designers mistakenly accepted user preference as the only feedback on the success of the total design. The antidote requires that you abstain from such automatic responses. Instead, use soul. It turns out that user performance gives a better answer for many design issues. However, sometimes user preference also provides good feedback when done correctly. This refinement in response constitutes soul testing. Let’s sort out the issues and show the special cases where user preference becomes useful.

GETTING USER FEEDBACK What good is testing? Testing provides you feedback. Feedback let’s you know how well you’re doing your job. We all ask “How am I doing?” in one way or another; in

sports, in relationships, in personal growth. Usability testing tells us how we’re doing on our design work. In our seminar Practical Usability Testing, we cover many points where usability testing counts during the development life cycle. And some (not all) of these points need user preference measures. We’ll cover them now, starting with the most general observation.

Marketplace Preferences In our examples above, we assumed that performance was primary, and preference a secondary criterion of success. For example, in corporate systems, you have a captive audience—users who must use the software to do their job. Luckily, most users willingly share your design goal of speed, accuracy, and ease of learning. Together, these performance goals contribute to user “performance.” Note that users cannot assert “preference” to reject a system that is fast, accurate, and easy to learn. Otherwise they would appear impractical to their managers. Also, you can explain the design tradeoffs to these corporate users and mollify their preference concerns.

But what if your application must be sold to individual users? What is the criterion of satisfaction in the marketplace? It’s clearly a different situation than we have with a captive audience. The design must captivate attention and earn user allegiance by any legal means. Clearly, we now must appeal to preference more than performance. As the adage goes, “An unsold application is unusable!”

In such cases, collect both performance data and preference data to uncover any contradictions such as we illustrated above. Our recommendation? Use your data to enhance user preference in areas where it makes a difference in sales. Meanwhile, avoid any “killer problems” by examining the performance data as well. Improve performance while keeping your product at a high level of com-

petitive preference. Remember, most software reviewers look at performance as well as preference qualities.

Function Testing This soul event must occur early in your development life cycle. In fact, most so-called “user testing” really is a mistaken and belated form of function testing. That is, designers often complete their application or prototype, and then see if users will find all the functions they need! This represents the junk food cryptotesting criticized early in the article. However, soul function testing doesn’t need a prototype or any detailed design at all. The issues uncovered with function testing are far more broad and comprehensive, as follows:

Not all functions are desirable or usable. Imagine a drive-up window at a fast food restaurant with featured instructions in Braille. Sound strange? We have seen labels for a first aid kit near the elevators of a high-rise office building—in Braille. What did the designer have in mind? In another case, a sales company issued an on-line scheduling calendar for salespeople who were on the road without computers. Developers prefer different functions than users. Generally, developers (and user representatives) identify functions that exceed user requirements. Developers see complexity as a “friendly challenge.” Thus, functions tend to be esoteric, complex, powerful, error prone, system related, and yes, impressive. However, typical users seek functions that are simple, practical, and common. Granted, they may sometimes be “wild,” as well.

Given a list of functions, you can test their suitability with written a questionnaire or survey. Here is a list of four kinds of surveys and their suitability. Make sure you target subjects who truly represent potential users of your application. Match the range of job types. If managers and marketing people are not going to use the application, don’t mingle their responses with those of “real users.” You

can compare data between these groups to detect any mismatch of expectations. In all cases, compare strength of user preference with the costs to develop the function. Insure that you get good bang for the buck, preference-wise. See Figures 3 to 6 for the four types of function tests.

How Much Would You Pay?

- How much would you pay for Item 1?
 - a. \$0
 - b. \$1 - \$5
 - c. \$6 - \$10
 - d. \$11 - \$20
 - e. \$21 - \$50
 - f. \$51 - \$100
 - g. More than \$100
- How much would you pay for Item 2?
- Etc.

Figure 3. This is the best type of function test. It provide realistic economic "weighting" to decisions regarding which feature to keep over other features. However, this only works if the subject actually buys your type of software, is knowledgeable about the features, has a personal, meaningful budget, and has an economic limit. (Too much wealth lets the subject pick all features without penalty.)

Limited Funds: How Much Would You Spend On Each?

- Your development budget is limited to \$100,000.
- Please allocate the budget between the following items:
 1. Item One \$ _____
 2. Item Two \$ _____
 3. Item Three \$ _____
 4. Etc.

Figure 4. Second best. In this form of questionnaire, there is no natural limit to expenditures. You must impose a hypothetical limit. Can your subject avoid getting carried away?

Limited Number: Which Would You Pick?

Please check the 10 most important items.

- Item 1 _____
- Item 2 _____
- Item 3 _____
- ...
- Item 30 _____

Figure 5. Third best. Sometimes dollars don't make any sense in your situation. However, this method only indicates which function is at the top of the list. You don't get any fine tuning regarding the degree of differences. Using dollar amounts, as in the first two methods, gives you a sense of the relative weighting or desirability of the functions.

Testing Your Task Design You got the right functions. Then you figured out the task design. Did you create a prototype yet? We hope not! Task design must precede detailed screen design. How do you know you have a suitable UM architecture—the mental model that makes your application easy to understand? How do you know you have a reasonable division into modules? How do you know whether your sequence of task events suits your users? Have you brainstormed about problems and alternatives? Your task design provides another point in the development life cycle where you get your money's worth with usability testing. And your best input comes from your user population!

Use a group walkthrough for best results. Walkthrough testing with prospective users is inexpensive (comparatively) and you can set it up quickly. You get a clear evaluation of the task flow early in the design process. You can evaluate competing solutions before committing money and time to development.

Keep walkthroughs within 60 to 90 minutes. Start

Rate on a Numerical Scale

Please circle a number from 1 to 6 that best matches your opinion of the value of each of the following items:

Item 1:

Not valuable - 1 2 3 4 5 6 - Extremely valuable

Item 2:

Not valuable - 1 2 3 4 5 6 - Extremely valuable

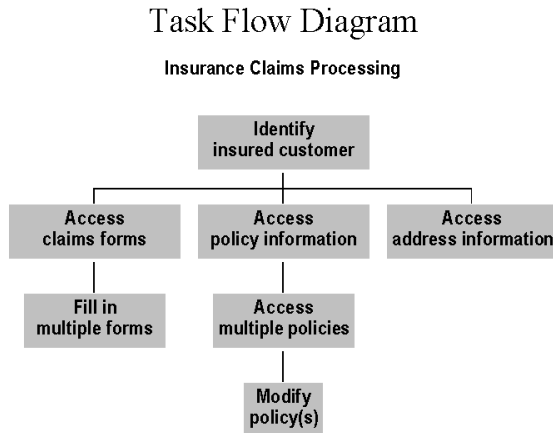
Item 3:

Etc.

Figure 6. Fourth best and really weak. Subjects can easily select every item as “Extremely Valuable” as a way out of making tough decisions. Take the results with a grain of salt. You may not really have a winner! Better used with a small number of subjects whom you can interview for their reasons.

with simple tasks to get the brainstorm juices flowing. Cover several core functions across a range of tasks. Make sure you get to areas that have raised concern elsewhere. Your goal is to get user feedback—not to defend yourself! Avoid drawn-out, exhausting battles of opinion. Invite a wide range of user types. As a brainstorming session, you need to hear a variety of viewpoints. It's okay to have supervisors, managers, as well as end-users with various job descriptions. However, if employees may feel intimidated around managers, use separate sessions. Create a storyboard to illustrate your task flow. Avoid detailed screen designs—they inhibit assessment of the big picture. Start with an outline of the task flow. Represent each step graphically for basic functions, basic navigation, and basic screen flow. Use specific examples. Be creative. Cartoons are OK! (See Figures 7 and 8.)

Run the walkthrough like a brainstorming session. Remember how to do brainstorms? Have a colleague keep notes on a flip chart. Let participants see progress so they don't return to old topics. Encourage alternatives and questions. Be flexible.



Task flow drives interface architecture!

Figure 7. Get discussion on your concept of the task. Only after you get agreement on a diagram such as this do you have the material from which to design a high level architecture, the first screen that connects everything together. (See our article in The X Journal, Nov-Dec, 1995.)

Be solution oriented within practical limits. Users perform a great job just pointing out problems. Don't get bogged down demanding solutions during the sessions. Keep things moving. Avoid brainstorming pitfalls. No negativity (it inhibits creative thought). No defensiveness (users do you a favor by criticizing). No ego involvement (good luck)! No technical jargon. Don't ask users how they would handle some detailed design. Keep to the big issues! Don't make decisions on the spot.

CONCLUSIONS When you run into a bad case of management cryptotestiness, now you can steer things in a positive direction. Apply soul to the

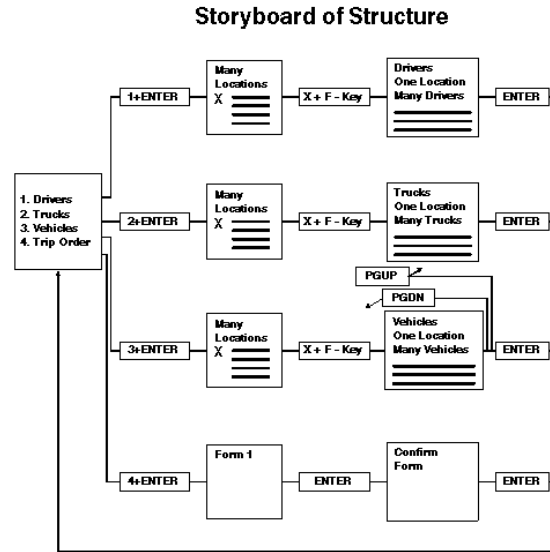


Figure 8. You can capture the task flow in a high level architecture with a “storyboard” like this. We use this picture in our exercise on improving high level architecture. It has a ton of problems. Your task design walkthrough would catch such problems! Include your development colleagues in walkthroughs that have design challenges. Users may not catch them. By the way, you often find problems yourself just by doing a storyboard early in the design process.

question of when to get user feedback (get it early). Steer your management towards early testing of functions and task design. Users excel in providing the critical feedback you need early in the development life cycle. (We said “early” three times!) Avoid at all costs the cryptopitfall of getting user preferences only at the end of your development—it’s tantamount to anosognosia—the “disease of knowledge.”