by NIALL MURPHY

<BACK

Designing User Interfaces: What Does the Customer Really Want?

In an ideal world, a Human Computer Interface professional would design the interactions that take place between the user and the machine. Unfortunately, we're not living in an ideal world.

Rupert Adley

Building a user interface is an onerous task. When specifying a product, the user interface is often the most complex part of the customer requirements. Part of the reason for this difficulty is that many of the other complex engineering issues are considered implementation issues, because the user may never be aware of them. For example, how much do you need to know about guiding laser beams when you press play on your CD player? By definition the user will be aware of everything on the user interface. In an ideal world the interface would be specified in part by a Human Computer Interface (HCI) professional. However not many of us are working in such an ideal world, and the engineer regularly has to take on this role, often with no formal training.

In the software world one cynical view is that many user interfaces start life as debuggers. How much of the internal state of the software is exposed to the outside world can depend on what the developer needs to know. A graphical front end is then tacked on to this debugger. Embedded developers have a slightly better reputation, partly because the discipline of industrial design has an influence on the hardware, and the software interface has to function through that hardware.

One of the reasons it is so hard to evaluate and refine the user interface is because it is so hard to measure. The number of messages a network router must process in a minute can be specified as a number, which marketing can demand and engineering can either deliver or negotiate—it is a quantity that all involved can understand. A bigger number is better but usually costs more. For the user interface,

Engineers are the ones under deadline pressure, and it is they who must force all changes to be justified.

however, the requirement given to an engineering team could vary from "the product shall be easy to use for a novice user" to a full specification describing the response to any given key pressed and defining the exact layout of a keypad or the screen layouts for a Graphical User Interface (GUI).

Neither the engineers nor the marketing team will be able to quantify the usability of the system and say "Yes, it is 75% usable for the novice and 82% usable for the experienced user." Not only that, but for a given change there is often disagreement between everyone concerned as to whether that change will improve usability or not. To add to the difficulty, some of these decisions are being made before there is a system on which to test any of the ideas. Even when a system is available to try out variations of the user interface, the kind of hard data that can be obtained in other areas of engineering will not be available, and the disputes will continue.

A feedback loop exists between engineering and marketing, even if those functions are performed by the same people in a small company. The engineers produce an interface, whether in paper form, slide show, mock-up, or full-functioning prototype. The reaction of the user to this interface is evaluated, which leads to changed requirements and further trals. This feedback loop generally decays over time as changes become minor and do not lead to further changes. Unfortunately, there is also the possibility that the changes that occur with each iteration are so major that they always lead to further changes. Often a change that is only intended to have a minor impact can upset the overall consistency of the interface, leading to compromises in unrelated areas. When this happens, the change should be reevaluated.

Damping this feedback loop so that it decays rapidly is the key to meeting deadlines. Usually the engineers are the ones under deadline pressure, and it is they who must force all changes to be justified or suffer the consequences.

One of the things to be wary of as requirements change is that change can be bad for the structure of the software. While the interface may be incrementally improved, each round of changes may compromise the internal design as the software is coerced into doing something that was not envisaged when the design began. All too often it is possible to implement the change with a hack, which may come back to haunt you. The coder rationalizes it by planning to return to code it properly once the interface is stable. I will leave it as an exercise to the reader to figure out how often that happens.

New versions of the interface could be tried out in a special build, with no changes committed to the master sources, unless it was decided that they were acceptable. This process does not work because all the new features have to be seen together, and some survive and others do not.

Some features make it through the first round only to be shot down during a future round of trials that are supposed to be examining an unrelated feature.

Another approach would be to do a complete rewrite of the user interface code once the interface is stable. Whether this option is feasible depends on the application and the time scales. The important point is that the engineer should be aware that this is an area of danger.

The rate of change of requirements in the early stages can be high simply because the interactions may be complex enough that there are flaws and inconsistencies in a specification that will not be discovered until it is being implemented. In one part of the specification it may say, "Under condition A turn on light X." In another part it may say, "Under condition B turn off light X." During implementation it may transpire that A and B can be simultaneously true.

Another cause of change is that when interactions are specified, designers may have to depend on their imagination. Seeing the real thing might not live up to the expectations.

The hardware aspects of the interface do not suffer as much from change as the software, especially later in the project when change is most destructive. The primary reason for this situation is money. Adding a feature to software in an embedded product does not add to the cost per unit, while adding a feature to hardware almost always will. There is often a tooling cost or external contractors or suppliers who are involved in the production of hardware, and changes to hardware will often lead to bills from these vendors. Changes in software costs in engineering time, but because engineering time does not appear on a purchase order, no one has to sign off for it, and it is not perceived as an extra cost. It is simply seen as an example of the software engineers being late again. Even if a conscious decision were made to compare costs that way, the external vendor cost would be a quote to which developers would more than likely stick, while the engineers asked to estimate the time to perform the change will invariably underestimate. Engineers may not allow for the time to write extra tests or the cost of unforeseen bugs in the new code.

SAMPLE USERS

One of the most important parts of the whole process of deciding what needs to be changed is observing users and interpreting their comments. Most of the texts on the subject of interface design advocate involving the user at all stages of the development cycle. This approach is necessary if the interface is the central component of the product, such as a flight simulator or virtual reality application. For applications with less demanding interfaces, a high level of user input can be disruptive. At the very start of the project, try to ensure that the engineers are exposed to the needs of the user so they learn how to look at their own work from the point of view of the user. This outlook will benefit them throughout the project.

Demonstrations are often performed by the engineer responsible for creating a significant part of the interface.

The cost of statistically significant surveys of user preferences is often prohibitive, unless the product is purely interface related, such as a new alternative to the mouse. Often user feedback comes from token users who are walked into the engineer's office, given a quick demo of the product, allowed to try it out for a while, and asked for an opinion.

Be careful in your choice of user. There is an adage in the psychology world that most published psychology is the psychology of college freshmen. This adage comes from the fact that the most convenient and plentiful supply of guinea pigs for academic researchers is the undergraduates in the researchers' own institution, which of course can skew any results. If you get the engineers working down the hall to try out your VCR interface, the complexity of programming timers may not be a challenge to them, but a broader group containing non-technical people might show great difficulty in this area.

Demonstrations are often performed by the engineer responsible for creating a significant part of the interface. A user may be less critical of the interface because someone is present who may feel personally criticized. It is often better if the designer is not present or at least does not play a central role in the discussions. If the team is not known to the user, then the designer can easily open with the line, "This is what the folks in engineering came up with, but I am not too sure about it." The demonstrator is distanced from the design, leaving the audience feeling less threatened. By being mildly critical of the interface, the presenter sets the tone to encourage others to express whatever they think without fear of offending. It may also be useful to point out to the user that the interface is being tested, not the user. This avoids situations in which the user may pretend to understand something to avoid appearing stupid. You also want to avoid the Hawthorne effect, which is peoples' tendency to work harder when they know they are being observed, which leads to non-typical behavior.

In the discussions that follow such demonstrations, it is important to have a strict agenda. When people are asked what they think of a certain interaction, they tend to say that X is wrong and then describe the interaction that would work. As the observer, it is important that you find out about X being wrong, and why. Do not let token users go too far down the road of describing the solution. The group may move to a discussion of a user's hypothetical interface, rather than the one that is being demonstrated. The solution presented by a user may be a valid one, and if so, the design team can think it through later and contact the user again for details. The reason that demonstrations are not good opportunities for gathering solutions is that often a specific example activity is chosen for the demonstration. The user will describe an interface that suits the particular example, but that may not represent a good general solution.

Use sample operators to find out two things: the needs of the user, and the faults in the current interface. Sometimes it is interesting to let users talk about whatever occurs to them as a good idea and sometimes politeness demands it. However, do not allow these ideas to dominate the exercise.

Is the customer always right? I may be committing marketing heresy by suggesting that the customer may sometimes be wrong. You will have to ask yourself the same question many times during preference trials. When a customer asks for a feature, it will often be up to you to point out the trade-offs. If a display shows a list of records in alphabetical order, the user may give reasons why a chronological ordering would be preferable. You as a designer know that there is going to be a performance penalty if the list has to be sorted each time it is going to be displayed, users may be barely satisfied with current speeds, and the performance penalty may make them decide against the change. Often, the engineer can see these issues ahead of time while a user could not be expected to foresee the tradeoffs. Performance versus usability will be one difficult tradeoff.

In safety critical applications there will be a tradeoff between usability and safety. The Therac-25 was a cancer irradiation device whose faulty operation led to a number of deaths. As a safety feature, all the settings for the device had to be entered through a terminal as well as on a control panel. This was considered redundant by operators of a prototype. It was "redundant" in the best sense of the word, but this feature was not appreciated by the users, who assumed that the safety of the equipment was beyond doubt. The design was changed before release so that the settings could be entered on the terminal alone. Once the settings were accepted by hitting the return key, the user was asked to confirm that the settings were those that were actually required. This confirmation was performed by pressing the return key again. This extra step was considered a replacement for the original second interface.

Unfortunately, users soon learned to press the return key twice in succession, because they knew that they would always be asked for confirmation. The two presses, similar to a double-click on a mouse, became a single action in the mind of the user, and no actual review of the settings was performed. Due to faulty software, some of the settings were occasionally not properly recorded. Since the cross check of having the settings entered twice had been removed, the fault was not detected. This was a case where what the user asked for was definitely not what the user needed.

In safety critical applications there will be a tradeoff between usability and safety.

Another tradeoff is power of expression versus ease of use. The most obvious example here is the Unix shell compared to a windowed interface with a file manager. The file manager allows us to select certain files and perform operations on them, such as moving or deleting, by manipulation with the mouse. The command line is more expressive. Regular expressions and wildcarding can be used to state exactly which files are to be deleted, and operations can be made recursive throughout a tree of directories. However, the command line is more difficult to use and is non-directed, in that the user is presented with no choices. The graphical interface is a more directed interface that lets users see a finite number of things in front of them to act on. In the early stages most users will be looking for a directed interface, but as they become more experienced, the desire for expressive power increases, and the tolerance of complexity increases. If you are constantly changing your trial users, they may stay at the novice stage. At the other extreme there is a danger in using the same users all the time. Users, like designers, are aware of the history of the development of the interface, and that history sometimes justifies idiosyncrasies a fresh user would find far less acceptable.

In the interests of getting early feedback from users, without the cost of building functioning interfaces, paper prototypes are sometimes used. These prototypes can take the shape of storyboard-type demonstrations where the user is presented with a sequence of pictures of the interface, as some operation is performed. While pen and paper make good brainstorming tools, this method of presenting an interface to users is extremely limited. A storyboard will always give the impression of a very slow interface. After being shown a four-key sequence, the user will often respond, "You mean it takes me that long just to initialize the system?" The user's requests will be based on reducing the number of steps rather than making the interface more intuitive. If the user actually pressed the four keys on a working interface it would take only a couple of seconds and not seem as laborious as turning a number of pages.

When a different picture is shown for each step, it can be hard for users to pick out exactly which parts of the picture have changed, and they are forced to scan the whole display each time. With a functioning prototype they are looking at one display, and changes catch the eye immediately. Note that the functioning interface may not represent a full system. Often, an impression of the final display can be presented in graphical format on a workstation. The important distinction is that the final display is functioning. In many cases, such an interface can serve as the front end for the production code for a large part of the real system.

IMPLEMENTATION

Some issues arise when you specify the interface. The overall consistency of the interface will often rest with the engineer, unless marketing has defined an exact specification. This situation is not necessarily a bad thing because the engineering disciple puts a lot of value on consistency. Consistency in the code is often reflected in the interface and vice versa, so the code can be a good place to detect inconsistency: "I can use this `blinkLight` routine everywhere except this one place. Why? There may be something different about what the user sees as well."

One of the things that can stand in the way of a consistent, orthogonal interface is compatibility. The competitors or predecessors of a particular piece of equipment may have set standards for the way certain operations are performed. For some common operations you may have to follow the established norm, even if that does not fit in with the way other operations are performed. Engineers often find this frustrating—their elegant design is being soiled by what is seen as an artificial and unfair requirement, created by history, rather than being part of the perfect solution to the problem at hand.

In many cases the interface of a modern piece of equipment will have complete software control of something that conventionally was an electro-mechanical control, such as on/off switches and volume control knobs on TV sets. Output devices, such as a needle to indicate fuel level in a car, could be changed from a mechanical devices to an electronic indicator that can be software controlled. This change opens up many opportunities that were not present before, such as expressing the remaining fuel as a percentage, or in gallons, or in number of miles remaining before the fuel runs out. In each case we allow the user to form a mental model of the fuel tank. Each of these models is different, but once we choose one we must be sure that we can fill in all the information that the model requires. To convert from gallons to miles remaining, we need a conversion factor. Using the average miles per gallon of the car may not be sufficient if the driver has been sitting in a traffic jam for 45 minutes. What seemed initially a very simple requirement now demands that the fuel monitor also monitors consumption. Now the user has a new rule to learn. The number of miles read from the monitor is only valid if the user continues the current driving pattern. You may think that this is a trivial rule to understand, but it is important to realize that every rule such as this that is created by the designer must eventually be learned by users if they are to make the most effective use of the interface. The user may learn this rule in the user manual, if this user is one of the few people who reads that section of the car's manual. Otherwise it is learned from careful observation, because the interface does not make this information explicit. Such hidden rules can be dangerous. If there are too many of them, users will constantly find themselves being surprised by the device's actions, and this will lead to mistrust.

Having moved from an analog device to a digital display, many interfaces fall into the trap of giving users far more precision than they require. There is no value in telling users that they have enough fuel for another 31.7 miles, since they would be foolish to wait until the last 0.7 of a mile before refueling. The vagueness of the needle in this case is an advantage. The needle says, "I do not know exactly how much further you can go, and we are not at the panic stage yet, but if you see a gas station, you may as well stop." Since no one except clairvoyants can know exactly how many more miles the fuel will last, this information may be more appropriate than "31.7" miles as an indicator, which may lure the driver into a false sense of security. I do not think many drivers would appreciate a display that said 31.7 miles ± 10%. Remember, they are not all engineers and may not get exposed to the concept of tolerance every day.

If the driver is low on fuel, the sensible thing to do is fill up at the next opportunity. The needle gauges are notoriously non-linear, but drivers rarely complain. The designer may do well to copy the needle approach and just display a bar graph made up of LEDs, which tells the driver enough to know when to refuel. This display is also less threatening for the new user, who can relate easily to past experience with the needle display. Sometimes copying the functionality of the original mechanical device is the simplest model for both the user and the designer.

While trying to manage all possible ways in which the user may use the interface to control the equipment or to request information, the handling of illegal input is sometimes neglected. The easiest route is to have one function that handles all invalid input with a beep and a generic prompt if text display is possible. The novice user's perception of user friendliness can be affected by the error handling because the novice is the person most likely to be making a large number of errors. Messages that state that the user has done something wrong may make the novice user feel intimidated. A message such as "Illegal key pressed" goes so far as to suggest that the user is a law breaker.

Making the error handling context sensitive can be a good investment. With a little care, prompts that guide the user onto a sensible path can replace "illegal key" messages. For example, on a tuning device the "store" key may be invalid unless some channel has been selected. The "illegal key" prompt could be replaced with "no channel selected." The user is now being supplied with a piece of information, beyond the fact that the wrong key was pressed.

In the medical field, it is not desirable that the beep, which indicates an invalid action, be too loud. The patient may be able to hear the machine, and high pitched beeps increase a patient's anxiety, even though the cause may be a harmless mistake, such as pressing an invalid key. This case is extreme, but users of other equipment may not appreciate announcing whenever they press the wrong key. At the least it discourages experimentation with the interface, and at worst the user will avoid the device completely.

Sometimes the problem of what response should be given to an invalid input can be solved by ensuring that at any time, any input the user can make is valid. One example is to replace a numeric keypad with a knob that is rotated to change value. With the keypad you would have to check that the number entered is not greater than the maximum allowed. However, with the knob the value is being changed in increments and the software can clip the value at the maximum and effectively decouple the knob from the display once that maximum is reached. Another way of avoiding invalid actions on a GUI is to remove on-screen buttons that are not currently valid.

I hope this advice will help a little bit in your struggles to design a better interface. It is always more difficult for engineers to deal with issues that are not perfectly measurable or mathematically definable. If you spend long enough working in this area, you will find yourself experimenting every time you take money from the automatic teller machine or watching family members try to figure out how to use the new VCR. Of course you have already hidden the user manual to make it a true test of the interface.

Niall Murphy is a software engineer working with Nellcor Puritan Bennett, Ireland, where he designs and develops user interfaces for medical equipment in C and C++. He has a degree in computer science from Trinity College, Dublin, and has contributed to several magazines. Murphy can be contacted via his e-mail address at nmurphy@iol.ie. ESP

REFERENCES

Leveson, Nancy, "*Safeware, System Safety and Computers*," Addison-Wesley Publishing, Reading, MA 1995.

Thimbleby, Harold, "*User Interface Design*," ACM Press, New York, New York, 1990.