

Lecture 16: Usability Engineering

Spring 2004

6.170 Laboratory in Software Engineering

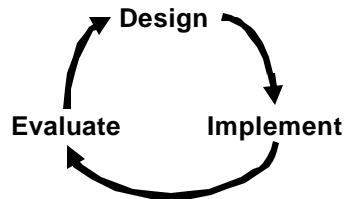
1

Outline

- Iterative design
- Low-fidelity prototypes
- Heuristic evaluation
- User testing

Iterative Design

- UI development is an iterative process



- Do you want the design cycle to be internal, or visible to your customers?

Spring 2004

6.170 Laboratory in Software Engineering

3

User interface development is inherently **risky**. UI design is hard for all the reasons we discussed in the last lecture. (You are not the user; the user is always right, except when the user isn't; users aren't designers either.) We don't (yet) have an easy way to predict how whether a UI design will succeed.

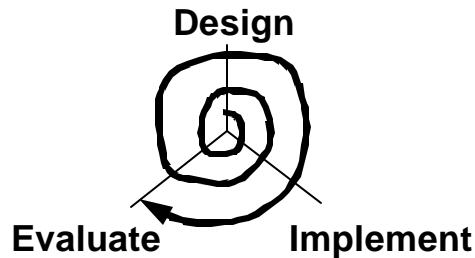
Iterative design offers a way to manage the inherent risk in user interface design. In iterative design, the software is refined by repeated trips around a design cycle: first imagining it (design), then realizing it physically (implementation), then testing it (evaluation).

Unfortunately, many commercial UI projects inflict iterative design on their paying customers. They design a bad user interface, implement it, and release it. Evaluation then takes place in the marketplace, as hapless customers buy their product and complain about it. Then they iterate the design process on version 2.

On the other hand, if you keep all your design iterations in-house, you may never release anything! It's very costly to do every iteration of a design with a high-quality implementation in a language like Java or C++ -- especially if you discover you have to throw away all that code because the design was bad.

Spiral Model

- Use throw-away prototypes and cheap evaluation for early iterations



Spring 2004

6.170 Laboratory in Software Engineering

4

The **spiral model** offers a way out of the dilemma. We build room for several iterations into our design process, and we do it by making the early iterations as cheap as possible.

The radial dimension of the spiral model corresponds to the **cost** of the iteration step – or, equivalently, its **fidelity** or **accuracy**. For example, an early implementation might be a paper sketch or mockup. It's low-fidelity, only a pale shadow of what it would look and behave like as interactive software. But it's incredibly cheap to make, and we can evaluate it by showing it to users and asking them questions about it.

Low-fidelity Prototypes

- Paper is the fastest prototyping tool
 - Sketch windows, menus, dialogs, widgets
 - Crank out lots of designs and evaluate them
- Hand-sketching is OK – even preferable
 - Focus on behavior & interaction, not fonts & colors
- Paper prototypes can even be executed
 - Cut out windows and dialogs
 - Simulate the computer's responses by moving pieces around and writing on them

Spring 2004

6.170 Laboratory in Software Engineering

5

It turns out that **paper** is a terrific prototyping tool. If you sit down and write Java code for your UI without having drawn a sketch first, you're letting Java design the UI for you. **Always sketch it on paper first.**

Hand sketching is valuable because it focuses attention on the issues that matter in early design without distracting anybody with details. When you're sketching by hand, you aren't bothered with details like font, color, alignment, whitespace, etc. In a drawing program, you would be faced with all these decisions, and you might spend a lot of time on them – time that would clearly be wasted if you have to throw away this design. Hand sketching also improves the feedback you get when you show your design to other people. They're less likely to nitpick about details that aren't relevant at this stage. They won't complain about the color scheme if there isn't one. More important, however, a hand-sketch design seems less finished, less set in stone, and more open to suggestions and improvements. Architects have known about this phenomenon for many years. If they show clean CAD drawings to their clients in the early design discussions, the clients are less able to discuss needs and requirements that may require radical changes in the design. In fact, many CAD tools have an option for rendering drawings with a "sketchy" look for precisely this reason.

It turns out that paper designs can even be executed. A **paper prototype** is a physical mockup of the interface, mostly made of paper. It's usually hand-sketched on multiple pieces, with different pieces showing different menus, dialog boxes, or window elements. A paper prototype can be brought to life by a person who simulates what the computer would do in response to a user's "clicks" and "keystrokes", by rearranging pieces, writing custom responses, and occasionally announcing some effects verbally that are too hard to show on paper. Because a paper prototype is actually interactive, you can actually user-test it: give users a task to do and watch how they do it.

Usability Guidelines (“Heuristics”)

- Plenty to choose from
 - Nielsen’s 10 principles
 - Tognazzini’s 16 principles
 - Platform guidelines
- Uses
 - Help designers choose design alternatives
 - Help evaluators find problems in interfaces (“heuristic evaluation”)

Spring 2004

6.170 Laboratory in Software Engineering

6

Let’s look at some design guidelines now. **Usability guidelines, or heuristics**, are rules that distill out the principles of effective user interfaces. There are plenty of sets of guidelines to choose from – sometimes it seems like every usability researcher has their own set of heuristics. Most of these guidelines overlap in important ways, however. The experts don’t disagree about what constitutes good UI. They just disagree about how to organize what we know into a small set of operational rules.

For the basis of this lecture, we’ll use Jakob Nielsen’s 10 heuristics, which can be found on his web site (reference on last slide). Another good list is **Tog’s First Principles** (find it in Google).

Platform-specific guidelines are also important and useful to follow. Platform guidelines tend to be very specific, e.g. you should have a File menu, and there should be a command called Exit on it (not Quit, not Leave, not Go Away). Following platform guidelines ensures consistency among different applications running on the same platform, which is valuable for novice and frequent users alike. However, platform guidelines are relatively limited in scope, offering solutions for only a few of the design decisions in a typical UI.

Heuristics can be used in two ways: during design, to choose among different alternatives; and during evaluation, to find and justify problems in interfaces.

1. Match the Real World

- Use common words, not techie jargon
 - But use domain-specific terms where appropriate



Source: Interface Hall of Shame

- Don't put limits on user-defined names
- Allow aliases/synonyms in command languages
- Metaphors are useful but may mislead

Spring 2004

6.170 Laboratory in Software Engineering

7

Let's start by looking at each of Nielsen's 10 heuristics in detail.

First, the system should match the real world of the user's experience as much as possible. Nielsen's original name for this heuristic was "Speak the user's language", which is a good slogan to remember. If the user speaks English, then the interface should also speak English, not Geekish. Technical jargon should be avoided. Use of jargon reflects aspects of the system model creeping up into the interface model, unnecessarily. How might a user interpret the dialog box shown here? One poor user actually read *type* as a verb, and dutifully typed M-I-S-M-A-T-C-H every time this dialog appeared. The user's reaction makes perfect sense when you remember that most computer users do just that, *type*, all day. But most programmers wouldn't even think of reading the message that way. Yet another example showing that You Are Not The User.

Technical jargon should only be used when it is specific to the application domain and the expected users are domain experts. An interface designed for doctors shouldn't dumb down medical terms.

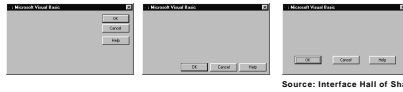
If an interface allows users to name things, then users should be free to choose long, descriptive names. Artificial limits on length or content should be avoided. DOS used to have a strong limit on filenames, an 8 character name and a 3 character extension. Echoes of these limits persist in Windows even today.

When designing an interface that requires the user to type in commands or search keywords, support as many aliases or synonyms as you can. Different users rarely agree on the same name for an object or command. One study found that the probability that two users would mention the same name was only 7-18%. (Furnas et al, "The vocabulary problem in human-system communication," *CACM* v30 n11, Nov. 1987).

Metaphors are one way you can bring the real world into your interface. A well-chosen, well-executed metaphor can be quite effective and appealing, but be aware that metaphors can also mislead. A computer interface must deviate from the metaphor at *some* point -- otherwise, why aren't you just using the physical object instead? At those deviation points, the metaphor may do more harm than good. For example, it's easy to say "a word processor is like a typewriter," but you shouldn't really *use* it like a typewriter. Pressing Enter every time the cursor gets close to the right margin, as a typewriter demands, would wreak havoc with the word processor's automatic word-wrapping.

2. Consistency and Standards

- Principle of Least Surprise
 - Similar things should look and act similar
 - Different things should look different
- Other properties
 - Size, location, color, wording, ordering, ...



- Command/argument order
 - Prefix vs. postfix
- Follow platform standards

Spring 2004

6.170 Laboratory in Software Engineering

8

The second heuristic is Consistency. This rule is often given the hifalutin' name the Principle of Least Surprise, which basically means that you shouldn't surprise the user with the way a command or interface object works. Similar things should look, and act, in similar ways. Conversely, different things should be visibly different.

Consistency is important to lots of properties. The screenshots show three different dialog box layouts used in Visual Basic. A consistent design would use only one.

A very important kind of consistency is in wording. Use the same terms throughout your user interface. If your interface says "share price" in one place, "stock price" in another, and "stock quote" in a third, users will wonder whether these are three different things you're talking about.

Incidentally, we've only looked at two heuristics, but already we have a contradiction! Matching the Real World argued for synonyms and aliases, so a command language should include not only *delete* but *erase* and *remove* too. But Consistency argues for only one name for each command, or else users will wonder whether these are three different commands that do different things. One way around the impasse is to look at the context in which you're applying the heuristic. When the *user* is talking, the interface should make a maximum effort to understand the user, allowing synonyms and aliases. When the *interface* is speaking, it should be consistent, always using the same name to describe the same command or object. What if the interface is smart enough to adapt to the user – should it then favor matching its output to the user's vocabulary (and possibly the user's inconsistency) rather than enforcing its own consistency? Perhaps, but adaptive interfaces are still an active area of research, and not much is known.

Command & argument ordering is another kind of consistency. In **noun-verb order**, the conventional order in graphical user interfaces, the user first selects the object of the command, and then invokes the command. In **verb-noun order**, the command is invoked first, and then the arguments are selected. A drawing program in which some commands were noun-verb and others were verb-noun would be very hard to learn and use.

There are three kinds of consistency you need to worry about: **internal consistency** within your application (like the VB dialog boxes shown); **external consistency** with other applications on the same platform (how do other Windows apps lay out OK and Cancel?); and **metaphorical consistency** with your interface metaphor or similar real-world objects.

Designs that are seriously inconsistent but provide only a tiny improvement in performance will probably fail. The Dvorak keyboard, for example, is slightly faster than the standard QWERTY keyboard, but not enough to overcome the power of an entrenched standard.

3. Help and Documentation

- Users don't read manuals
 - Prefer to spend time working toward their task goals, not learning about your system
- But manuals and online help are vital
 - Usually when user is frustrated or in crisis
- Help should be:
 - Searchable
 - Context-sensitive
 - Task-oriented
 - Concrete
 - Short

Spring 2004

6.170 Laboratory in Software Engineering

9

The next heuristic is (good) Help and Documentation. The sad fact about documentation is that most users simply don't read it, at least not before they try the interface. As a result, when they finally *do* want to look at the manual, it's because they've gotten stuck. Good help should take this into account.

4. User Control and Freedom

- Provide undo
- Long operations should be cancelable
- All dialogs should have a cancel button



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

10

This heuristic used to be called “Clearly Marked Exits” in Nielsen’s old list. Users should not be trapped by the interface. Every dialog box should have a cancel button (where is it in this CuteFTP dialog box?), and long operations should be interruptible.

Users should be able to explore the interface without fear of being trapped in a corner. Undo is a great way to support exploration.

5. Visibility of System Status

- Keep user informed of system state
 - Cursor change
 - Selection highlight
 - Status bar
 - Don't overdo it...
- Response time
 - < 0.1 s: seems instantaneous
 - 0.1-1 s: user notices, but no feedback needed
 - 1-5 s: display busy cursor
 - > 1-5 s: display progress bar



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

11

This heuristic used to be called, simply, “Feedback.” Keep the user informed about what’s going on. We’ve developed lots of idioms for feedback in graphical user interfaces. Use them:

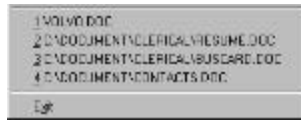
- Change the cursor to indicate possible actions (e.g. hand over a hyperlink), modes (e.g. drag/drop), and activity (hourglass).
- Use highlights to show selected objects. Don’t leave selections implicit.
- Use the status bar for messages and progress indicators.

But don’t overdo it. This dialog box demands a click from the user. Why? Does the interface need a pat on the back for finishing the conversion? It would be better to just skip on and show the resulting documentation.

Depending on how long an operation takes, you may need different amounts of feedback. Even though we say “no feedback needed” if the operation takes less than a second, remember that something should change, visibly, within 100 ms, or perceptual fusion will be disrupted.

6. Flexibility and Efficiency

- Provide shortcuts for frequent operations
 - Keyboard accelerators
 - Command abbreviations
 - Styles
 - Bookmarks
 - History



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

12

This heuristic used to be called “Shortcuts.” Frequent users need and want them. Recently-used history is one very useful kind of shortcut, like this recently-used files menu.

7. Error Prevention

- Selection is less error-prone than typing
 - But don't go overboard...



- Disable illegal commands
- Avoid modes

Spring 2004

6.170 Laboratory in Software Engineering

13

Now we get into heuristics about error handling. Since humans make errors if they're given a chance, the best solution is to prevent errors entirely. One way to prevent errors is to allow users to select rather type. Misspellings then become impossible. This attitude can be taken to an extreme, however, as shown in this example.

If a command is illegal in the current state of the interface – e.g., Copy is impossible if nothing is selected – then the command should be disabled (“grayed out”) so that it simply can't be selected in the first place.

Modes are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands.

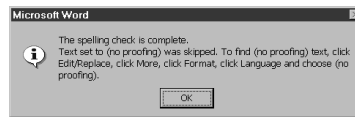
Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs.

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. When modes are necessary, it's essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn't actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user's locus of attention. That's why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn't really work. (Raskin, *The Humane Interface*, 2000 has a good discussion of locus of attention as it relates to mode visibility.)

Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do something active to stay in the alternate mode, essentially eliminating the chance that they'll forget what mode they're in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another spring-loaded mode; you're only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette, that drawing mode is active only for one mouse gesture. Once you've drawn one rectangle, the mode automatically reverts to ordinary pointer selection.

8. Recognition, Not Recall

- Use menus, not command languages
- Use combo boxes, not textboxes
- Use generic commands where possible (Open, Save, Copy Paste)
- All needed information should be visible



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

14

There's another reason why selection is better than typing – it reduces the user's memory load. "Minimize Memory Load" was the original name for this heuristic, and it drives much of modern user interface design.

Norman (in *The Design of Everyday Things*) makes a useful distinction between **knowledge in the head**, which is hard to get in there and still harder to recover, and **knowledge in the world**, which is far more accessible. Knowledge in the head is what we usually think of as knowledge and memory. Knowledge in the world, on the other hand, means not just documentation and button labels and signs, but also **nonverbal** features of a system that constrain our actions or remind us of what to do. Command languages demand lots of knowledge in the head, while menus rely on knowledge in the world.

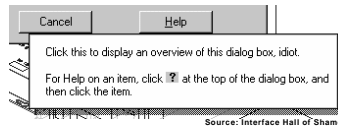
Generic commands are polymorphic, working the same way across a wide variety of data objects and applications. Generic commands are powerful because only one command has to be learned and remembered.

Any information needed by a task should be visible or otherwise accessible in the interface for that task. The interface shouldn't depend on users to *remember* the email address they want to send mail to, or the product code for the product they want to buy.

This dialog box is a great example of overreliance on the user's memory. It's a modal dialog box, so the user can't start following its instructions until after clicking OK. But then the instructions vanish from the screen, and the user is left to struggle to remember them. An obvious solution to this problem would be a button that simply executes the instructions directly! This message is clearly a last-minute patch for a usability problem.

9. Error Reporting, Diagnosis, and Recovery

- Be precise; restate user's input
 - Not "Cannot open file", but "Cannot open file named paper.doc"
- Give constructive help
 - why error occurred and how to fix it
- Be polite and nonblaming
 - Not "fatal error", not "illegal"
- Hide technical details (stack trace) until requested



Spring 2004

6.170 Laboratory in Software Engineering

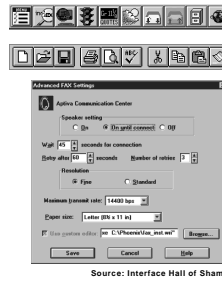
15

If you can't prevent the error, give a good error message. A good error message should (1) be precise; (2) speak the user's language, avoiding technical terms and details unless explicitly requested; (3) give constructive help; and (4) be polite. The message should be worded to take as much blame as possible away from the user and heap the blame instead on the system. Save the user's face; don't worry about the computer's. The computer doesn't feel it, and in many cases it is the interface's fault anyway for not finding a way to prevent the error in the first place.

The tooltip shown here came from a production version of AutoCad! As the story goes, it was inserted by a programmer as a joke, but somehow never removed before release.

10. Aesthetic and Minimalist Design

- “Less is More”
 - Omit extraneous info, graphics, features
- Good graphic design
 - Few, well-chosen colors and fonts
 - Follow color guidelines
 - Group with whitespace
 - Align controls sensibly
- Use concise language
 - Choose labels carefully



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

16

The final heuristic is a catch-all for a number of rules of good graphic design, which really boil down to one word: simplicity. Leave things out unless you have good reason to include them. Don't put more help text on your main window than what's really necessary. Leave out extraneous graphics. Most important, leave out unnecessary features. If a feature is never used, there's no reason for it to complicate your interface.

Use few, well-chosen colors. The toolbars at the top show the difference between cluttered and minimalist color design. The first toolbar is full of many saturated colors. It's not only gaudy and distracting, but actually hard to scan. The second toolbar, from Microsoft Office, uses only a handful of colors – black, white, gray, blue, yellow. It's muted, calming, and the few colors are used to great effect to distinguish the icons. The whitespace separating icon groups helps a lot too.

The dialog box shows how cluttered and incomprehensible a layout can look when controls aren't aligned.

Summarizing the Heuristics

- **Meet expectations**
 1. Match the real world
 2. Consistency & standards
 3. Help & documentation
- **User is the boss**
 4. User control & freedom
 5. Visibility of system status
 6. Flexibility & efficiency
- **Handle errors**
 7. Error prevention
 8. Recognition, not recall
 9. Error reporting, diagnosis, and recovery
- **Keep it simple**
 10. Aesthetic & minimalist design

Spring 2004

6.170 Laboratory in Software Engineering

17

Since it's hard to learn 10 heuristics and hold them in your head when you're trying to design, I find it useful to categorize Nielsen's heuristics still further.

Meet expectations. The first three heuristics concern how well the interface fits its environment, its task, and its users: speaking the user's language, keeping consistent with itself and other applications, and satisfying the expectation of help when it's needed.

User is the boss. The next three heuristics are related in that the interface should serve the user, rather than the other way around. Don't push the boss into the corner, keep the boss aware of things, and make the boss productive and efficient.

Handle errors. The next three heuristics largely concern errors, which are part and parcel of human-computer interaction: prevent them as much as possible, don't rely on human memory, but when errors are unavoidable, report them properly.

Aesthetic & minimal design stays in its own category, as befits its overwhelming importance.

Keep it simple.

Heuristic Evaluation

- Inspection technique
- Steps
 - Inspect UI thoroughly
 - Compare UI against heuristics
 - List usability problems
 - Justify every problem with a heuristic
 - “Too many choices on the home page – Aesthetic & Minimalist Design”
 - Can’t just say “I don’t like the colors”
 - Not limited to the 10 heuristics, however

Spring 2004

6.170 Laboratory in Software Engineering

18

One application of these 10 heuristics is a usability inspection process called **heuristic evaluation**. Heuristic evaluation was originally invented by Jakob Nielsen, and you can learn more about it on his web site. Nielsen has done a number of studies to evaluate the effectiveness of heuristic evaluation. Those studies have shown that heuristic evaluation’s cost-benefit ratio is quite favorable; the cost per problem of finding usability problems in an interface is generally cheaper than alternative methods.

Heuristic evaluation is an inspection method. It is performed by a usability expert – someone who knows and understands the heuristics we’ve just discussed, and has used and thought about lots of interfaces.

The basic steps are simple: the evaluator inspects the user interface thoroughly, judges the interface on the basis of the heuristics we’ve just discussed, and makes a list of the usability problems found – the ways in which individual elements of the interface deviate from the usability heuristics.

Your evaluation should be grounded in known usability guidelines. You should justify each problem you list by appealing to a heuristic, and explaining how the heuristic is violated. This practice helps remove most of the (inevitable) subjectivity involved in inspections: You can’t just say “that’s an ugly yellow color.” (If it’s really yucky, you *should* pass that subjective opinion back to the design team, but you’ll be forced to identify it as subjective if you can’t find a heuristic to justify it.) But you don’t have to limit yourself to the Nielsen 10. Tog’s principles are also good, as are platform guidelines (although they also fall into Nielsen’s Consistency and Standards heuristic). Last lecture, we saw a number of specific usability principles that can serve equally well: Fitts’s Law, perceptual fusion, color guidelines. The Nielsen 10 are helpful in that they’re a short list that covers a wide spectrum of usability problems. For each element of the interface, you can quickly look down the Nielsen list to guide your thinking.

Outline

- Iterative design
- Low-fidelity prototypes
- Heuristic evaluation
- **User testing**

User Testing

- Start with a prototype
- Write up a few representative tasks
 - Short, but not trivial
 - e.g.: “add this meeting to calendar”, “type this letter and print it”
- Find a few representative users
 - 3 is often enough to find obvious problems
- Watch them do tasks with the prototype

Spring 2004

6.170 Laboratory in Software Engineering

20

User testing is the gold standard for evaluating a user interface. Since it’s hard to predict how a typical user will respond to an interface, the best way to learn is to actually find some typical users, put them in front of your interface, and watch what happens.

You don’t need to have a finished implementation to do user testing. A paper prototype is enough to test, and it’s so easy to build (relative to code) that paper prototypes are often the first version of your interface that you test on users.

A good user test shouldn’t be undirected. Don’t just plop a user down and say “try this interface”. You should prepare some representative tasks that are appropriate to your interface. Pick tasks that are common, tasks that should be easy, and tasks that you’re worried may be hard. Make the tasks short (if possible), but not trivial. Make each task concrete (e.g., “schedule a meeting for 3pm this Wednesday”), but don’t provide specific instructions on how to do it.

Once you have your tasks, find some users that are representative of your target user population. Needless to say, don’t use people from the development team, even if they happen to fall in the target user population. They know too much about the underlying system, so they’re not typical users. A handful of users is usually enough for feedback about obvious usability problems. (If you wanted to measure some quantitative improvement due to your design, however, you’d need many more users, and you’d need to carefully design the testing.)

How to Watch Users

- Brief the user first (being a test user is stressful)
 - “I’m testing the system, not testing you”
 - “If you have trouble, it’s the system’s fault”
 - “Feel free to quit at any time”
 - Ethical issues: informed consent
- Ask user to think aloud
- Be quiet!
 - Don’t help, don’t explain, don’t point out mistakes
 - Sit on your hands if it helps
 - Two exceptions: prod user to think aloud (“what are you thinking now?”), and move on to next task when stuck
- Take lots of notes

Spring 2004

6.170 Laboratory in Software Engineering

21

Once you have your tasks and your users, the final step is simple: **watch what happens**. This is harder than it sounds.

First, being a test user is stressful for most people. There’s a tendency to feel like a subject of an intelligence test. If they can’t figure out how to use your interface, they may feel like they’ve failed. You need to be aware of this phenomenon, and take steps in advance to ward it off. When you brief a user before a test, make very clear that the goal of the test is to uncover problems in the computer program. **Anything that goes wrong is the interface’s fault, not the user’s**. Assure them that they can quit the test at any time.

User studies conducted in connection with MIT research should also be cognizant of the ethical issues surrounding use of human subjects. MIT policies treat the user of humans in software user studies identically with their use in psychology experiments, drug trials, and studies of new medical procedures. You have to obtain approval for a user study from MIT’s Committee on the Use of Humans as Experimental Subjects (COUHES).

While the user is actually using your interface, encourage them to **think aloud**: verbalize what they’re thinking as they use the interface. Encourage them to say things like “OK, now I’m looking for the place to set the font size, usually it’s on the toolbar, nope, hmm, maybe the Format menu...” Thinking aloud gives you (the observer) a window into their thought processes, so you can understand what they’re trying to do and what they expect. Thinking aloud can be hard to do, particularly when the user gets absorbed in the task. Sometimes you have to nudge the user a little: “what are you thinking now?” “Why did you look there?”

While the user is talking, you, as the observer, should be doing the opposite: **keeping quiet**. Don’t offer any help, don’t attempt to explain the interface. Just sit on your hands, bite your tongue, and watch. You’re trying to get a glimpse of how a typical user will interact with the interface. Since a typical user won’t have the system’s designer sitting next to them, you have to minimize your effect on the situation. It may be very hard for you to sit and watch someone struggle with a task, when the solution seems *obvious* to you, but that’s how you learn the usability problems in your interface.

You have only two excuses for opening your mouth during a user test: first, to prod the user to think aloud, and second, to move the user along to another task if they really get stuck.

Keep yourself busy by taking a lot of notes.

Watch for Critical Incidents

- Critical incidents: events that strongly affect task performance or satisfaction
- Usually negative
 - Errors
 - Repeated attempts
 - Curses
- Can also be positive
 - “Cool!”
 - “Oh, now I see.”

Spring 2004

6.170 Laboratory in Software Engineering

22

What should you take notes about? As much as you can, but focus particularly on **critical incidents**, which are moments that strongly affect usability, either in task performance (efficiency or error rate) or in the user’s satisfaction. Most critical incidents are negative. Pressing the wrong button is a critical incident. So is repeatedly trying the same feature to accomplish a task. Users may draw attention to the critical incidents with their think-aloud, with comments like “why did it do that?” or “@%#@#%!” Critical incidents can also be positive, of course. You should note down these pleasant surprises too.

Critical incidents give you a list of potential usability problems that you should focus on in the next round of iterative design.

Summary

- You are not the user
- Keep human capabilities in mind
 - Perception, motor, memory, color
- Iterate over your design
- Make cheap, throw-away prototypes
- Evaluate them with heuristics and users

Further Reading

- General books on usability
 - Johnson. *GUI Bloopers: Don'ts and Dos for Software Developers and Web Designers*, Morgan Kaufmann, 2000.
 - Jef Raskin, *The Humane Interface*, Addison-Wesley 2000.
 - Hix & Hartson, *Developing User Interfaces*, Wiley 1995.
- Low-fidelity prototyping
 - Rettig, "Prototyping for Tiny Fingers", CACM April 1994.
- Usability heuristics
 - Nielsen, "Heuristic Evaluation."
<http://www.useit.com/papers/heuristic/>
 - Tognazzini, "First Principles."
<http://www.asktog.com/basics/firstPrinciples.html>