Lecture 5:
Design Principles

UI Hall of Fame or Shame?

Source: Interface Hall of Shame

IBM's RealCD is CD player software, which allows you to play an audio CD in your CD-ROM drive.

Why is it called "Real"? Because its designers based it on a real-world object: a plastic CD case. This interface has a *metaphor*, an analogue in the real world. Metaphors are one way to make an interface "intuitive," since users can make guesses about how it will work based on what they already know about the interface's metaphor. Unfortunately, the designers' careful adherence to this metaphor produced some remarkable effects, none of them good.

Here's how RealCD looks when it first starts up. Notice that the UI is dominated by artwork, just like the outside of a CD case is dominated by the cover art. That big RealCD logo is just that – static artwork. Clicking on it does nothing.

There's an obvious problem with the choice of metaphor, of course: a CD case doesn't actually play CDs. The designers had to find a place for the player controls – which, remember, serve the primary task of the interface – so they arrayed them vertically along the case hinge. The metaphor is dictating control layout, against all other considerations.

Slavish adherence to the metaphor also drove the designers to disregard all consistency with other desktop applications. Where is this window's close box? How do I shut it down? You might be able to guess, but is it "intuitive?" Learnability comes from more than just metaphor.

UI Hall of Shame!

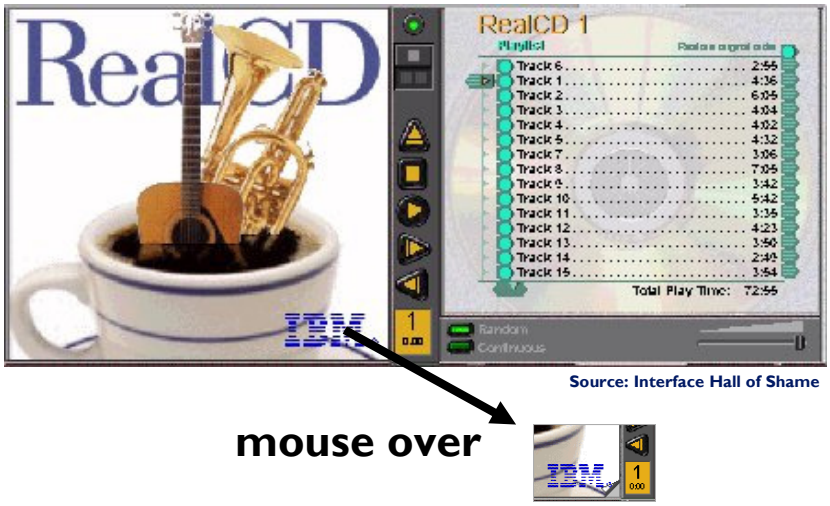
Source: Interface Hall of Shame

But it gets worse. It turns out, like a CD case, this interface can also be opened. Oddly, the designers failed to sensibly implement their metaphor here. Clicking on the cover art would be a perfectly sensible way to open the case, and not hard to discover once you get frustrated and start clicking everywhere. Instead, it turns out the only way to open the case is by a toggle button control (the button with two little gray squares on it).

Opening the case reveals some important controls, including the list of tracks on the CD, a volume control, and buttons for random or looping play. Evidently the metaphor dictated that the track list belongs on the "back" of the case. But why is the cover art more important than these controls? A task analysis would clearly show that adjusting the volume or picking a particular track matters more than viewing the cover art.

And again, the designers ignore consistency with other desktop applications. It turns out that not all the tracks on the CD are visible in the list. Could you tell right away? Where is its scrollbar?

UI Hall of Shame



Source: Interface Hall of Shame

mouse over

Fall 2006 6.831 UI Design and Implementation 4

We're not done yet. Where is the online help for this interface?

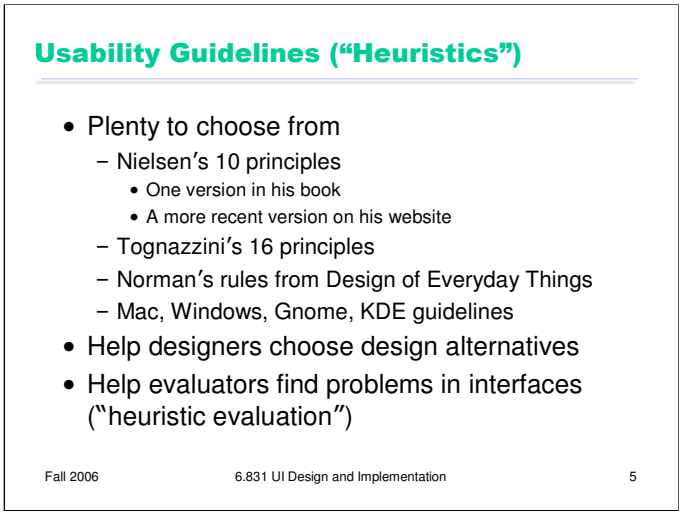
First, the CD case must be open. You had to figure out how to do that yourself, without help.

With the case open, if you move the mouse over the lower right corner of the cover art, around the IBM logo, you'll see some feedback. The corner of the page will seem to peel back. Clicking on that corner will open the Help Browser.

The aspect of the metaphor in play here is the *liner notes* included in a CD case. Removing the liner notes booklet from a physical CD case is indeed a fiddly operation, and alas, the designers of RealCD have managed to replicate that part of the experience pretty accurately. But in a physical CD case, the liner notes usually contain lyrics or credits or goofy pictures of the band, which aren't at all important to the primary task of playing the music. RealCD puts the *instructions* in this invisible, nearly unreachable, and probably undiscoverable booklet.

This example has several lessons: first, that interface metaphors can be horribly misused; and second, that the presence of a metaphor does not at all guarantee an "intuitive", or easy-to-learn, user interface. (There's a third lesson too, unrelated to metaphor – that beautiful graphic design doesn't equal usability, and that graphic designers can be just as blind to usability problems as programmers can.)

Fortunately, metaphor is not the only way to achieve learnability. In fact, it's probably the hardest way, fraught with the most pitfalls for the designer. In this lecture, we'll look at some other ways.



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. (An older version of the same heuristics, with different names but similar content, can be found in his *Usability Engineering* book, one of the recommended books for this course.) Another good list is **Tog's First Principles** (find it in Google), 16 principles from Bruce Tognazzini that include affordances and Fitts's Law. In the last lecture, we talked about some design guidelines proposed by Norman: visibility, affordances, constraints, feedback, and so on.

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 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.

Guidelines From Earlier Lectures

- User-centered design
  - Know your users
  - Understand their tasks
- Fitts's Law
  - Size and proximity of controls should relate to their importance
  - Tiny controls are hard to hit
  - Screen edges are precious
- Memory
  - Use chunking to simplify information presentation
  - Minimize working memory
- Color guidelines
  - Don't depend solely on color distinctions (color blindness)
  - Avoid red on blue text (chromatic aberration)
  - Avoid small blue details

Here are some of the guidelines we've already discussed in earlier lectures.

1. Match the Real World

- Use common words, not techie jargon
  - But use domain-specific terms where appropriate
- Don't put limits on user-defined names
- Allow aliases/synonyms in command languages
- Metaphors are useful but may mislead

Source: Interface Hall of Shame

Let's look 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

- Examples
  - Desktop
  - Trashcan
- Dangers of metaphors
  - Often hard for designers to find
  - Deceptive
  - Constraining
  - Breaking the metaphor
- Use of a metaphor doesn't excuse other bad design decisions

Metaphors are one way you can bring the real world into your interface. We started out by talking about RealCD's, an example of an interface that uses a strong metaphor in its 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.

Let's close by looking again at metaphors.

The advantage of metaphor is that you're borrowing a conceptual model that the user already has experience with. A metaphor can convey a lot of knowledge about the interface model all at once. It's *a notebook.* It's a *CD case*. It's a *desktop*. It's a *trashcan*. Each of these metaphors carries along with it a lot of knowledge about the parts, their purposes, and their interactions, which the user can draw on to make guesses about how the interface will work.

Some interface metaphors are famous and largely successful. The desktop metaphor – documents, folders, and overlapping paper-like windows on a desk-like surface – is widely used and copied. The trashcan, a place for discarding things but also for digging around and bringing them back, is another effective metaphor – so much so that Apple defended its trashcan with a lawsuit, and imitators are forced to use a different look. (Recycle Bin, anyone?)

The basic rule for metaphors is: use it if you have one, but don't stretch for one if you don't. Appropriate metaphors can be very hard to find, particularly with real-world objects. The designers of RealCD stretched hard to use their CD-case metaphor (since in the real world, CD cases don't even play CDs), and it didn't work well.

Metaphors can also be deceptive, leading users to infer behavior that your interface doesn't provide. Sure, it looks like a book, but can I write in the margin? Can I rip out a page?

Metaphors can also be constraining. Strict adherence to the desktop metaphor wouldn't scale, because documents would always be full-size like they are in the real world, and folders wouldn't be able to have arbitrarily deep nesting.

The biggest problem with metaphorical design is that your interface is presumably more capable than the real-world object, so at some point you have to break the metaphor. Nobody would use a word processor if *really* behaved like a typewriter. Features like automatic word-wrapping break the typewriter metaphor, by creating a distinction between hard carriage returns and soft returns.

Most of all, using a metaphor doesn't save an interface that does a bad job communicating itself to the user. Although RealCD's model was metaphorical – it opened like a CD case, and it had a liner notes booklet inside the cover – these features had such poor visibility and perceived affordances that they were ineffective.

Direct Manipulation

- User interacts with visual representation of data objects
  - Continuous visual representation
  - Physical actions or labeled button presses
  - Rapid, incremental, reversible, immediately visible effects
- Examples
  - Files and folders on a desktop
  - Scrollbar
  - Dragging to resize a rectangle
  - Selecting text
- Visual representation and physical interaction are important

Direct manipulation is the preeminent interface style for graphical user interfaces. It's often used for metaphorical interfaces, because it simulates the way we interact with real physical objects. Direct manipulation is defined by three principles [Shneiderman, *Designing the User Interface*, 2004]:

1. A **continuous visual representation** of the system's data objects. Examples of this visual representation include: icons representing files and folders on your desktop; graphical objects in a drawing editor; text in a word processor; email messages in your inbox. The representation may be verbal (words) or iconic (pictures), but it's continuously displayed, not displayed on demand. Contrast that with the behavior of *ed*, a command-language-style text editor: *ed* only displayed the text file you were editing when you gave it an explicit command to do so.

2. The user interacts with the visual representation using **physical actions** or **labeled button presses**. Physical actions might include clicking on an object to select it, dragging it to move it, or dragging a selection handle to resize it. Physical actions are the *most* direct kind of actions in direct manipulation – you're interacting with the virtual objects in a way that feels like you're pushing them around directly. But not every interface function can be easily mapped to a physical action (e.g., converting text to boldface), so we also allow for "command" actions triggered by pressing a button – but the button should be visually rendered in the interface, so that pressing it is analogous to pressing a physical button.

3. The effects of actions should be **rapid** (within 100 ms), **incremental** (you can drag the scrollbar thumb a little or a lot, and you see each incremental change), **reversible** (you can undo your operation – with physical actions this is usually as easy as moving your hand back to the original place, but with labeled buttons you typically need an Undo command), and **immediately visible**.

Direct manipulation so powerful because exploits the perceptual and motor skills of the human machine. Choosing the visual representation and physical interaction are the hard design problems here.

Direct Manipulation Design Principles

- Affordances
- Natural mapping
- Visibility
- Feedback

Don Norman, in his great book The Design of Everyday Things, identified a number of design principles from our interaction with physical objects, like doors and scissors. Since a direct manipulation interface is intended to be a visual metaphor for physical interaction, we'll look at some of these ideas and how they apply to computer interfaces.

Affordances

- Perceived and actual properties of a thing that determine how the thing could be used
  - Chair is for sitting
  - Knob is for turning
  - Button is for pushing
  - Listbox is for selection
  - Scrollbar is for continuous scrolling or panning
- Perceived vs. actual

According to Norman, affordance refers to "the perceived and actual properties of a thing", primarily the properties that determine how the thing could be operated.

Note that **perceived** affordance is not the same as **actual** affordance. A facsimile of a chair made of papier-mache has a perceived affordance for sitting, but it doesn't actually afford sitting: it collapses under your weight. Conversely, a fire hydrant has no perceived affordance for sitting, since it lacks a flat, human-width horizontal surface, but it actually does afford sitting, albeit uncomfortably.

The parts of a user interface should agree in perceived and actual affordances.

Natural Mapping

- Physical arrangement of controls should match arrangement of function
- Best mapping is direct, but natural mappings don't have to be direct
  - Light switches
  - Stove burners
  - Turn signals
  - Audio mixer

Another important principle is **natural mapping** of functions to controls.

Consider the spatial arrangement of a light switch panel. How does each switch correspond to the light it controls? If the switches are arranged in the same fashion as the lights themselves, it is much easier to learn which switch controls which light.

Direct mappings are not always easy to achieve, since a control may be oriented differently from the function it controls. Light switches are mounted vertically, on a wall; the lights themselves are mounted horizontally, on a ceiling. So the switch arrangement may not correspond *directly* to a light arrangement.

Other good examples of mapping include:

•Stove burners. Many stoves have four burners arranged in a square, and four control knobs arranged in a row. Which knobs control which burners? Most stoves don't make any attempt to provide a natural mapping.

•Car turn signals. The turn signal switch in most cars is a stalk that moves up and down, but the function it controls is a signal for left or right turn. So the mapping is not direct, but it is nevertheless natural. Why?

•An audio mixer for DJs (proposed by Max Van Kleek for the Hall of Fame) has two sets of identical controls, one for each turntable being mixed. The mixer is designed to sit in between the turntables, so that the left controls affect the turntable to the left of the mixer, and the right controls affect the turntable to the right. The mapping here is direct.

Visibility

- Relevant parts of system should be visible
  - Not usually a problem in the real world
  - But takes extra effort in computer interfaces

Visibility is an essential principle – probably the most important – in communicating a user interface to the user.

If the user can't *see* an important control, they would have to (1) guess that it exists, and (2) guess where it is. Recall that this was exactly the problem with RealCD's online help facility. There was no visible clue that the help system existed in the first place, and no perceivable affordance for getting into it.

Visibility is not usually a problem with physical objects, because you can usually tell its parts just by looking at it. Look at a bicycle, or a pair of scissors, and you can readily identify the pieces that make it work. Although parts of physical objects can be made hidden or invisible – for example, a door with no obvious latch or handle – in most cases it takes more design work to hide the parts than just to leave them visible.

The opposite is true in computer interfaces. A window can interpret mouse clicks anywhere in its boundaries in arbitrary ways. The input need not be related at all to what is being displayed. In fact, it takes more effort to make the parts of a computer interface visible than to leave them invisible. So you have to guard carefully against invisibility of parts in computer interfaces.

Feedback

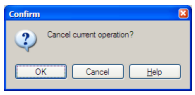
- Actions should have immediate, visible effects
  - Push buttons
  - Scrollbars
  - Drag & drop
- Kinds of feedback
  - Visual
  - Audio
  - Haptic

The final principle of interface communication is feedback: what the system does when you perform an action. When the user successfully makes a part work, it should appear to respond. Push buttons depress and release. Scrollbar thumbs move. Dragged objects follow the cursor.

Feedback doesn't always have to be visual. **Audio** feedback – like the clicks that a keyboard makes – is another form. So is **haptic** feedback, conveyed by the sense of touch. The mouse button gives you haptic feedback in your finger when you feel the vibration of the click. That's much better feedback then you get from a touchscreen, which doesn't give you any physical sense when you've pressed it hard enough to register.

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



Source: Interface Hall of Shame

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.

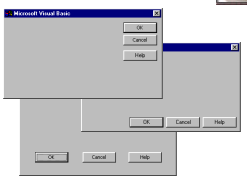
A very important kind of consistency is in wording. Use the same terms throughout your user interface. If your interface says "share price" in one place, "stock price" in another, and "stock quote" in a third, users will wonder whether these are three different things you're talking about. Conversely, use different terms when you mean different things. In WinSCP, when you cancel a long transaction, it pops up a seriously confusing dialog box that uses Cancel to mean two different things. It would be far better to label the buttons Yes and No here, to avoid the inconsistency of Cancel. (Thanks to Daniel Wendel for this example.)

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.

Kinds of Consistency

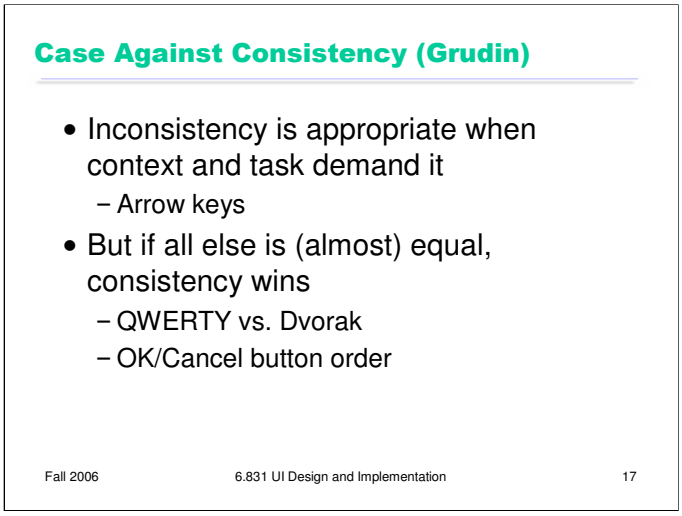
- Internal
- External
- Metaphorical



Fall 2006 6.831 UI Design and Implementation 16

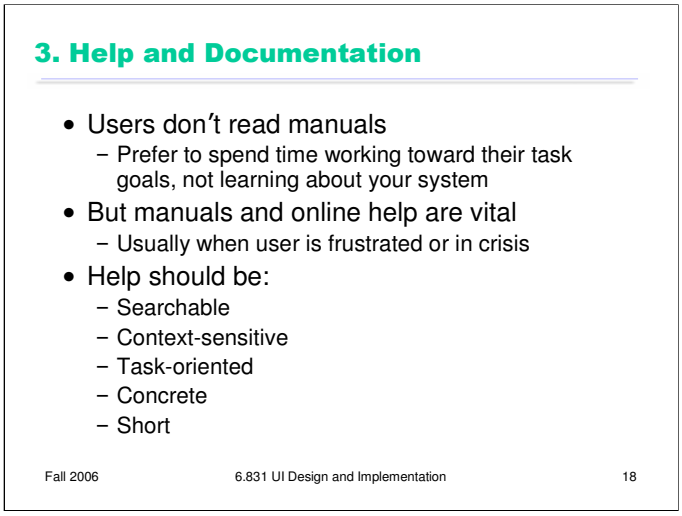
There are three kinds of consistency you need to worry about: **internal consistency** within your application (like the VB dialog boxes shown); **external consistency** with other applications on the same platform (how do other Windows apps lay out OK and Cancel?); and **metaphorical consistency** with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

We discussed the RealCD interface in an earlier lecture – it has problems with both metaphorical consistency (CD jewel cases don't play; you don't open them by pressing a button on the spine; and they don't open as shown), and with external consistency (the player controls aren't arranged horizontally as they're usually seen; and the track list doesn't use the same scrollbar that other applications do).



Jonathan Grudin (in "The Case Against User Interface Consistency, *CACM* v32 n10, Oct 1989) finesses the issue of consistency still further. His argument is that consistency should not be treated as a sacred cow, but rather remain subservient to the needs of context and task. For example, although the inverted-T arrow-key arrangement on modern keyboards is both internally and metaphorically inconsistent in the placement of the down arrow, it's the right choice for efficiency of use. If two design alternatives are otherwise equivalent, however, consistency should carry the day.

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.

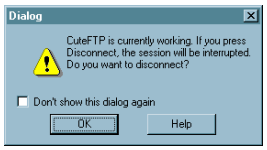


The next heuristic is (good) Help and Documentation. The sad fact about documentation is that most users simply don't read it, at least not before they try the interface. As a result, when they finally *do* want to look at the manual, it's because they've gotten stuck. Good help should take this into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

A good point was raised in class that exclusively task-oriented help (which has largely taken over in Microsoft Windows) makes it impossible to get a high-level overview of an interface from the manual. So it's possible to go too far.

4. User Control and Freedom

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



Source: Interface Hall of Shame

- User-provided data should be editable

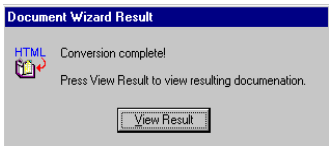
This heuristic used to be called "Clearly Marked Exits" in Nielsen's old list. Users should not be trapped by the interface. Every dialog box should have a cancel button (where is it in this CuteFTP dialog box?), and long operations should be interruptible.

Users should be able to explore the interface without fear of being trapped in a corner. Undo is a great way to support exploration, but it's not the only way. **Editing** is important too. If the user is asked to provide any kind of data – whether it's the name of an object, a list of email attachments, or the position of a rectangle – the interface should provide a way to go back and change what the user originally entered – rename the object, add or remove attachments, move around that rectangle some more. Data that is initialized by the user but can never again be touched will frustrate user control and freedom.

Providing user control and freedom can have strong effects on your backend model. You'll have to worry about undo, and you'll have to make sure things are mutable. If you built your backend assuming that a user-provided piece of data would never change once it had been created, you're going to have trouble building a good UI.

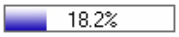
5. Visibility of System Status (Feedback)

- Keep user informed of system state
  - Cursor change
  - Selection highlight
  - Status bar
  - Don't overdo it...



Source: Interface Hall of Shame

- 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



Fall 2006 6.831 UI Design and Implementation 20

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.

Selections are particularly important. When the user selects some object and then operates on it with a command, **keep it selected**, especially if it changes appearance drastically or moves somewhere else. If the selected thing is offscreen when the user finally invokes a command on it, scroll it back into view. That allows the user to follow what happened to the object, so they can easily evaluate its final state. Similarly, if the user makes a selection and then invokes an unrelated command (like scrolling or sorting or filtering, none of which actually use the selection), **preserve the selection**, even if it means you have to remember it and regenerate it. User selections, like user data, are precious, and contribute to the visibility of what the system is doing.

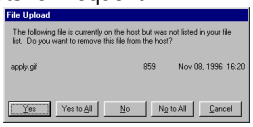
5. Visibility of System Status (Feedback)

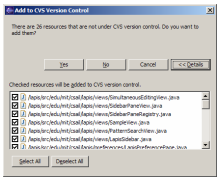
The Athena login screen provides no feedback at all when you enter your password. There's a security explanation for that – it doesn't want to leak any information about your password to somebody reading the screen over your shoulder – but it probably goes too far. A new user might think the interface is broken.

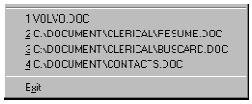
Most password entry fields provide at least *some* feedback; the convention is a row of asterisks. The bottom dialog, from Lotus Notes, provides even more feedback: a set of hieroglyphics derived from the password you typed in. If you can remember what hieroglyphics you usually see, then you can check them to see if you typed your password correctly, before pressing OK.

6. Flexibility and Efficiency (Shortcuts)

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







Source: Interface Hall of Shame

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.

We looked at some other shortcuts in an earlier hall of fame & shame. Yes to All and No to All were good, but they don’t smoothly handle the case where the user wants to choose a mix of Yes and No. Eclipse’s list of checkboxes, with Select All and Deselect All, provides the right mix of flexibility and efficiency.

7. Error Prevention

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

Source: Interface Hall of Shame

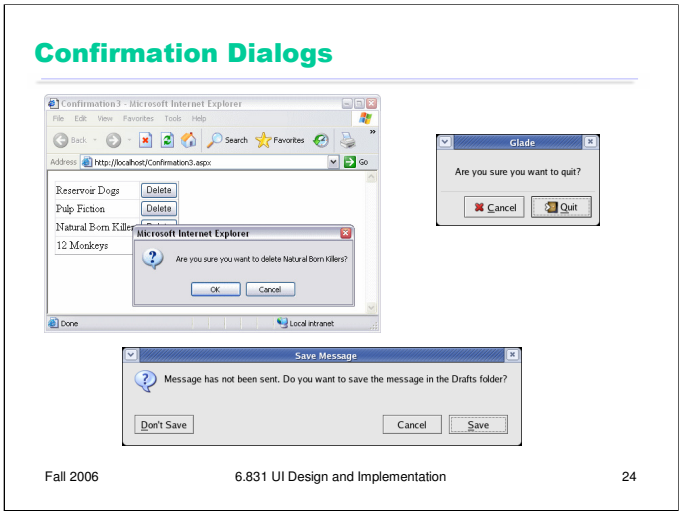
- Disable illegal commands
- Keep dangerous commands away from common ones

Now we get into heuristics about error handling. Since humans make errors if they're given a chance (this is called Murphy's Law: "if something can go wrong, it will"), 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.

You can also reduce errors by making sure that dangerous functions (hard to recover from if invoked accidentally) are well-separated from frequently-used commands. Outlook 2003 makes this mistake: when you right-click on an email attachment, you get a menu that mixes common commands (Open, Save As) with less common and less recoverable ones – if you print that big file by mistake, you can't get the paper back. And if you Remove the attachment, it's even worse – undo won't bring it back! (Thanks to Amir Karger for this example.)



An unfortunately common strategy for error prevention is the **confirmation dialog**, or "Are you sure?" dialog. It's not a good approach, and should be used only sparingly, for several reasons:

•Confirmation dialogs can substantially reduce the efficiency of the interface. In the example above, a confirmation dialog pops up whenever the user deletes something, forcing the user to make two button presses for every delete, instead of just one. Frequent commands should avoid confirmations.

•If a confirmation dialog is frequently seen – for example, every time the Delete button is pressed – then the expert users will learn to expect it, and will start to **chunk** it as part of the operation. In other words, to delete something, the user will learn to push Delete and then OK, without reading or even thinking about the confirmation dialog! The dialog has then completely lost its effectiveness, serving only to slow down the interface without actually preventing any errors.

In general, reversibility (i.e. **undo**) is a far better solution than confirmation. Even a web interface can provide at least single-level undo (undoing the last operation). Operations that are very hard to reverse may deserve confirmation, however. For example, quitting an application with unsaved work is hard to undo – but a well-designed application could make even this undoable, using automatic save or keeping unsaved drafts in a special directory.

Description Error

- Intended action is replaced by another action with many features in common
  - Pouring orange juice into your cereal
  - Putting the wrong lid on a bowl
  - Throwing shirt into toilet instead of hamper
  - Going to Kendall Square instead of Kenmore Square
- Avoid actions with very similar descriptions
  - Long rows of identical switches
  - Adjacent menu items that look similar

A description error occurs when two actions are very similar. The user intends to do one action, but accidentally substitutes the other. A classic example of a description error is reaching into the refrigerator for a carton of milk, but instead picking up a carton of orange juice and pouring it into your cereal. The actions for pouring milk in cereal and pouring juice in a glass are nearly identical – open fridge, pick up half-gallon carton, open it, pour– but the user's mental description of the action to execute has substituted the orange juice for the milk.

Description errors can be fought off by applying the converse of the Consistency heuristic: different things should look and act different, so that it will be harder to make description errors between them. Avoid actions with very similar descriptions, like long rows of identical switches.

Capture Error

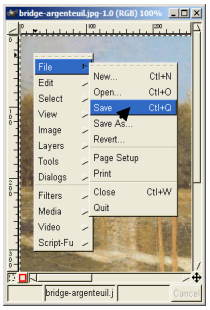
- A sequence of actions is replaced by another sequence that starts the same way
  - Leave your house and find yourself walking to school instead of where you meant to go
  - Vi :wq command
- Avoid habitual action sequences with common prefixes

A capture error occurs when a person starts executing one sequence of actions, but then veers off into another (often more familiar) sequence that happened to start the same way. A good mental picture for this is that you've developed a mental groove from executing the same sequence of actions repeatedly, and this groove tends to capture other sequences that start the same way.

In a computer interface, you can deal with capture errors by avoiding habitual action sequences that have common prefixes.

Mode Error

- Modes: states in which actions have different meanings
  - Vi's insert mode vs. command mode
  - Caps Lock
  - Drawing palette
- Avoiding mode errors
  - Eliminate modes
  - Visibility of mode
  - Spring-loaded or temporary modes
  - Disjoint action sets in different modes



A third kind of error is a mode error. **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. We talked about another mode error in Gimp: accidentally changing a menu shortcut because your mouse is hovering over it.

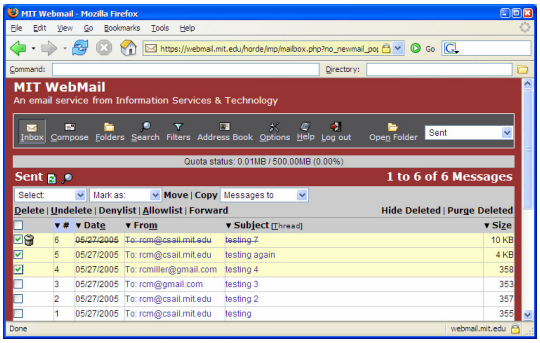
Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs.

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.

Finally, you can also mitigate the effects of mode errors by designing action sets so that no two modes share any actions. Mode errors may still occur, when the user invokes an action in the wrong mode, but the action can simply be ignored rather than triggering any undesired effect. (Although then, you might ask, why have two modes in the first place?)

More Mode Errors



The combination of Mozilla Firefox and MIT Webmail have some rather tricky modes involving keyboard shortcuts. The Alt-D shortcut does different things depending on what mode you're in:

- if you're browsing the web with Firefox, Alt-D puts the keyboard focus on the address bar, so you can type a URL.

- but if you're looking at a folder in MIT Webmail, Alt-D deletes the messages you've selected.

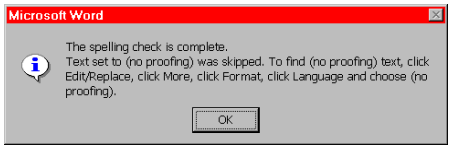
- if you're looking at a message in MIT Webmail, Alt-D normally deletes the message – which at least is consistent with the folder view.

- but if you're looking at an **already deleted** message in MIT Webmail, then the Delete command is missing – and Alt-D now invokes the Denylist command – which adds the sender of this message to a list of people whose messages get filtered out.

(Thanks to InHan Kang for this example.)

8. Recognition, Not Recall (Memory Load)

- 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

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. Affordances, constraints, and feedback are all aspects of knowledge in the world. Command languages demand lots of knowledge in the head, while menus rely on knowledge in the world.

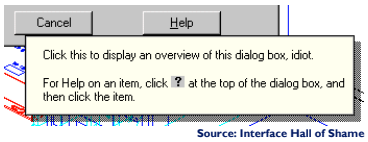
Generic commands are polymorphic, working the same way across a wide variety of data objects and applications. Generic commands are powerful because only one command has to be learned and remembered.

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, 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



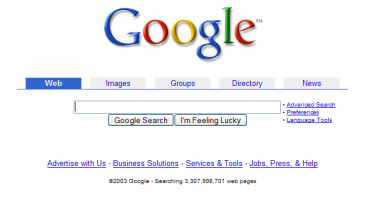
Source: Interface Hall of Shame

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 at the bottom 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



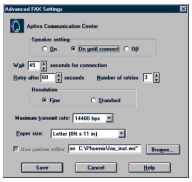
The final heuristic is a catch-all for a number of rules of good graphic design, which really boil down to one word: simplicity. Leave things out unless you have good reason to include them. Don't put more help text on your main window than what's really necessary. Leave out extraneous graphics. Most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

Google and the Tivo remote offer great positive examples of the less-is-more philosophy.

10. Aesthetic and Minimalist Design

- Good graphic design
  - Few, well-chosen colors and fonts

  - Group with whitespace
  - Align controls sensibly
- Use concise language
  - Choose labels carefully

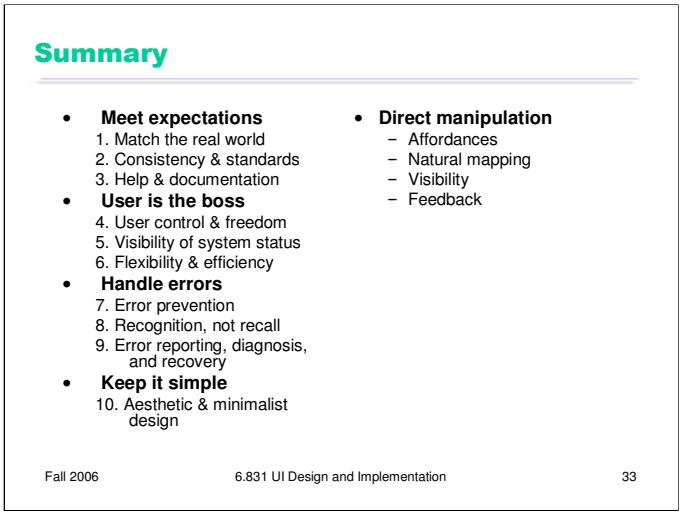


Source: Interface Hall of Shame

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.

We'll look at graphic design in more detail in a future lecture.



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.

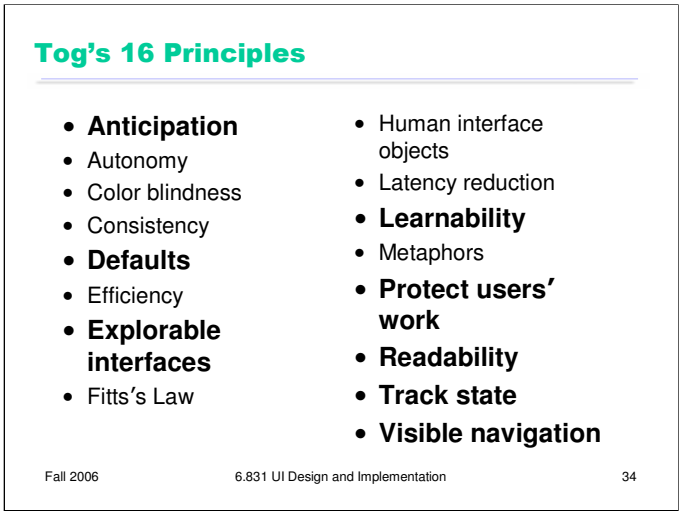
Meet expectations. The first three heuristics concern how well the interface fits its environment, its task, and its users: speaking the user's language, keeping consistent with itself and other applications, and satisfying the expectation of help when it's needed.

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.**

The **direct manipulation** rules are also important enough for their own category.



Let's look at a couple other lists of guidelines, because they highlight other good rules of design. Here is Bruce Tognazzini's list. We've seen most of these already; let's just focus on the few that are new, or that highlight particularly important problems to avoid.

Anticipation means that a good design should put all needed information and tools within the user's easy reach. Anticipation is the reason why a File Save dialog box needs a way to create a new folder. Note that you can't anticipate the user's needs without a thorough task analysis!

Defaults are common answers already filled into a form. Defaults help in lots of ways: they provide shortcuts to both novices and frequent users; they help the user learn the interface by showing examples of legal entries. But Tog advises that defaults should be *fragile*, coming up already selected so that frequent users can immediately overtype them. Tog also advises removing the actual word "default" from your interface's vocabulary, which makes sense because it has some very negative connotations in the lending world.

Explorable interfaces is basically User Control and Freedom, but deserves special notice. One way users learn is by exploring: poking around an interface, trying things out. Your interface should encourage this kind of exploration, with physically reversible actions, undo, and cancel. For example, users navigating around a 3D world or a complex web site can easily get lost; give them an easy, obvious way to get back to some "home", or default view.

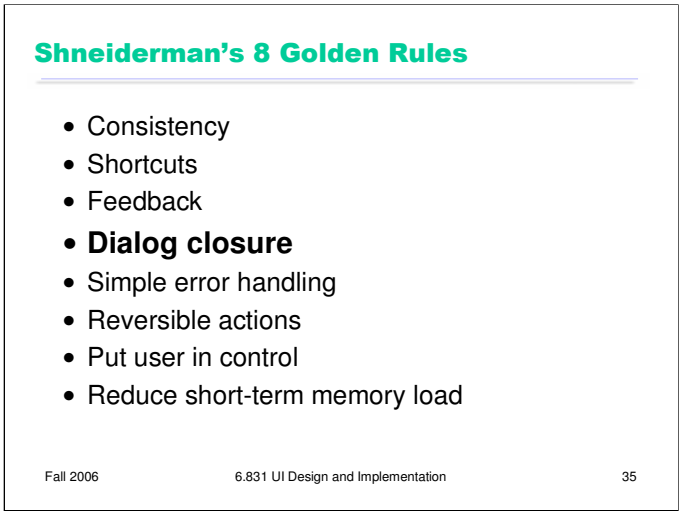
Learnability is one of our usability criteria (along with efficiency, memorability, error rate, and satisfaction). *Every* user is a novice with your interface at some point. Design for learnability. Even if your target users are frequent users, who receive heavy direct training in using your interface, you can still make design decisions that make this learning easier.

Protect users' work is certainly error prevention, but it highlights an important value judgment: errors that lose or destroy the user's work are the worst kind. It's worth substantial engineering to prevent this from happening.

Readability is a graphic design question, but it's one of the most important aspects of graphic design in most user interfaces, whether desktop or web. Choose font size and color contrast to maximize the readability of text, particularly for aging users.

Track state is a kind of shortcut in the sense that you should save the user from restoring the state of their last session. Keep histories of users' previous choices; when you run the Print function again, remember the settings the user provided before.

Finally, **visible navigation** is a kind of visibility of system state. On the Web, in particular, users are in danger of getting lost. Help prevent this by visualizing the user's location; popular techniques include bread crumb trails (like Business & Economy >> Finance & Investing >> Banking) and highlights in navigation bars.



One more list: Shneiderman's 8 Golden Rules of UI design include most of the principles we've already discussed. The new one is **dialog closure**. Action sequences should be designed with a beginning, a middle, and an end. For example, think about drag and drop:

At the beginning, you press the mouse button and see the object picked up with your cursor.

In the middle, you move the object across the screen towards your target, getting feedback that it's coming along.

At the end, you release the mouse button, and see the effects of the drop.

The key feature of closure is the feedback you get at the **end** of the operation. This assurance that the operation completed provides the user with a sense of accomplishment, some relief, and an opportunity to clear their working memory of the details of the task in preparation for another.