

Usability

Prof. Rob Miller

MIT EECS

IAP 2010

6.470 IAP Web Programming Competition

User Interface Hall of Shame



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

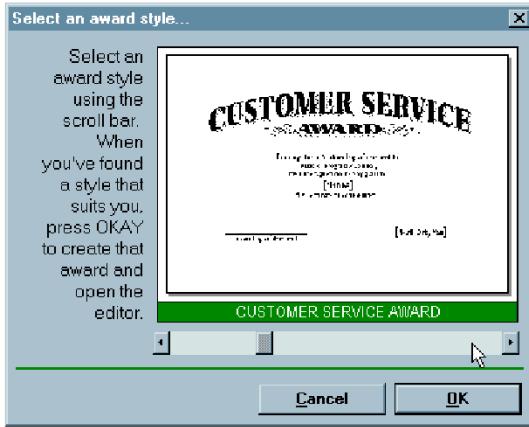
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template for the certificate.

This interface is clearly graphical. It's mouse-driven – no memorizing or typing complicated commands. It's even what-you-see-is-what-you-get (WYSIWYG) – the user gets a preview of the award that will be created. So why isn't it usable?

The first clue that there might be a problem here is the long help message on the left side. Why so much help for a simple selection task? Because the interface is bizarre! The *scrollbar* is used to select an award template. Each position on the scrollbar represents a template, and moving the scrollbar back and forth changes the template shown.

This is a cute but bad use of a scrollbar. Notice that the scrollbar doesn't have any marks on it. How many templates are there? How are they sorted? How far do you have to move the scrollbar to select the next one? You can't even guess from this interface.

User Interface Hall of Shame



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

Normally, a horizontal scrollbar underneath an image (or document, or some other content) is designed for scrolling the content horizontally. A new or infrequent user looking at the window sees the scrollbar, assumes it serves that function, and ignores it. **Inconsistency** with prior experience and other applications tends to trip up new or infrequent users.

Another way to put it is that the horizontal scrollbar is an **affordance** for continuous scrolling, not for discrete selection. We see affordances out in the real world, too; a door knob says “turn me”, a handle says “pull me”. We’ve all seen those apparently-pullable door handles with a little sign that says “Push”; and many of us have had the embarrassing experience of trying to pull on the door before we notice the sign. The help text on this dialog box is filling the same role here.

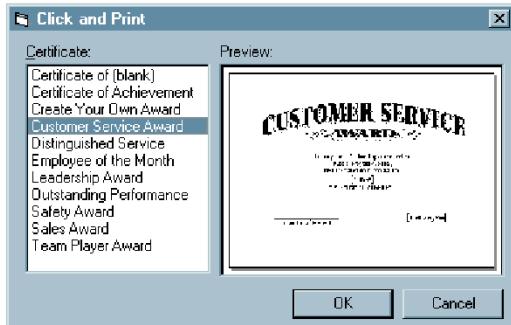
But the dialog doesn’t get any better for frequent users, either. If a frequent user wants a template they’ve used before, how can they find it? Surely they’ll remember that it’s 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes what should be a random access process and transforms it into a linear process. Every user has to look through all the choices, even if they already know which one they want. The computer scientist in you should cringe at that algorithm.

Even the help text has usability problems. “Press OKAY”? Where is that? And why does the message have a ragged left margin? You don’t see ragged left too often in newspapers and magazine layout, and there’s a good reason.

On the plus side, the designer of this dialog box at least recognized that there was a problem – hence the help message. But the help message is indicative of a flawed approach to usability. Usability can’t be left until the end of software development, like package artwork or an installer. It can’t be patched here and there with extra messages or more documentation. It must be part of the process, so that usability bugs can be *fixed*, instead of merely patched.

How could this dialog box be redesigned to solve some of these problems?

Redesigning the Interface



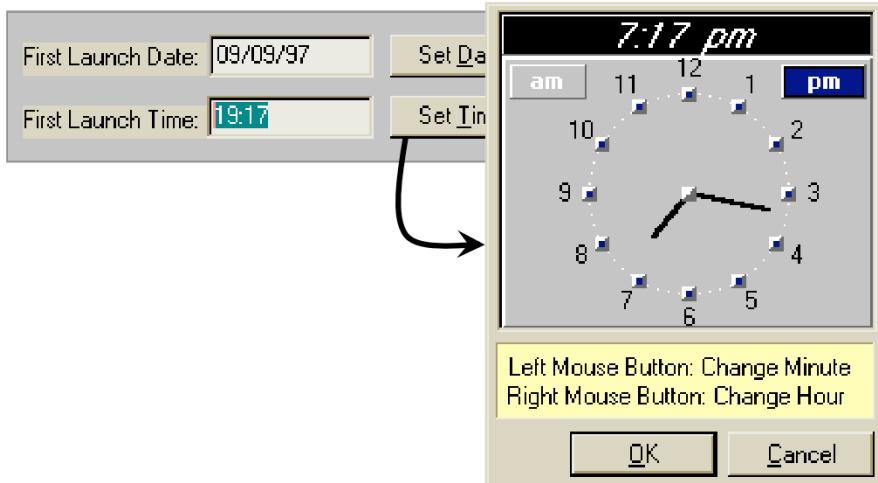
Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

Here's one way it might be redesigned. The templates now fill a list box on the left; selecting a template shows its preview on the right. This interface suffers from none of the problems of its predecessor: list boxes clearly afford selection to new or infrequent users; random access is trivial for frequent users. And no help message is needed.

Another for the Hall of Shame



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

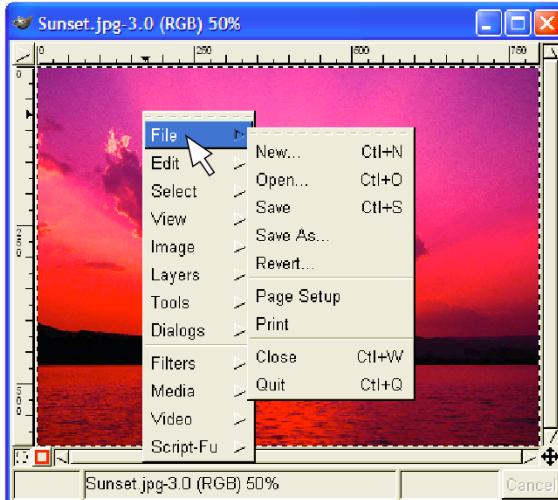
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to bring up a dialog box.

This dialog box displays time differently, using 12-hour time (7:17 pm) where the original dialog used 24-hour time (consistency!). Just to increase the confusion, it also adds a third representation, an analog clock face.

So how is the time actually changed? By clicking mouse buttons: clicking the left mouse button increases the minute by 1 (wrapping around from 59 to 0), and clicking the right mouse button increases the hour. Sound familiar? This designer has managed to turn a sophisticated graphical user interface, full of windows, buttons, and widgets, and controlled by a hundred-key keyboard and two-button mouse, into a **clock radio!**

Perhaps the worst part of this example is that it's not a result of laziness. Somebody went to a lot of effort to draw that clock face with hands. If only they'd spent some of that time thinking about usability instead.

Hall of Fame or Hall of Shame?



IAP 2010

6.470 IAP Web Programming Competition

Gimp is an open-source image editing program, comparable to Adobe Photoshop. Gimp's designers made a strange choice for its menus. Gimp windows have no menu bar. Instead, all Gimp menus are accessed from a *context menu*, which pops up on right-click.

This is certainly inconsistent with other applications, and new users are likely to stumble trying to find, for example, the File menu, which never appears on a context menu in other applications. (I certainly stumbled as a new user of Gimp.) But Gimp's designers were probably thinking about expert users when they made this decision. A context menu should be faster to invoke, since you don't have to move the mouse up to the menu bar. A context menu can be popped up anywhere. So it should be faster. Right?

Wrong. We'll see why later in this lecture.

Usability Defined

Usability = how well users can use the system's functionality

Dimensions of usability

- Learnability: is it easy to learn and remember?
- Visibility: is the state of the system visible?
- Efficiency: once learned, is it fast to use?
- Errors: are errors few and recoverable?
- Satisfaction: is it enjoyable to use?

IAP 2010

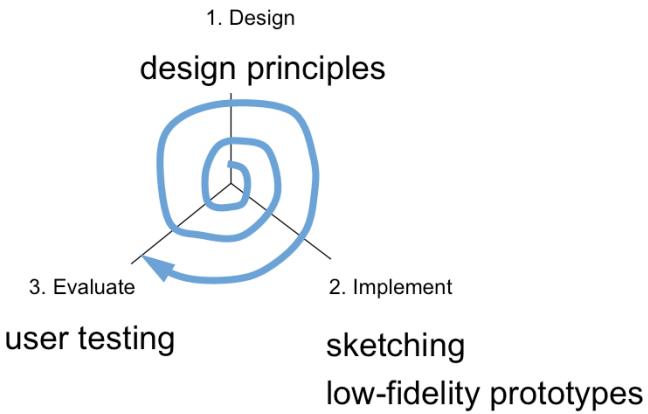
6.470 IAP Web Programming Competition

The property we're concerned with here, **usability**, is more precise than just how "good" the system is. A system can be good or bad in many ways. If important requirements are unsatisfied by the system, that's probably a deficiency in functionality, not in usability. If the system is very expensive or crashes frequently, those problems certainly detract from the user's experience, but we don't need user testing to tell us that.

More narrowly defined, usability measures how well users can use the system's functionality. Usability has several dimensions: learnability, visibility, efficiency, error rate/severity, and subjective satisfaction, among others.

Notice that we can **quantify** all these measures of usability. Just as we can say algorithm X is faster than algorithm Y on some workload, we can say that interface X is more learnable, or more efficient, or causes fewer errors than interface Y for some set of tasks and some class of users.

Lecture Outline



IAP 2010

6.470 IAP Web Programming Competition

Today we're going to talk about some ideas and techniques used in user interface design: (1) **design principles** that can guide your conception of a user interface; (2) **low-fidelity prototyping** techniques that help you try out your design cheaply and easily; and (3) **user testing** to measure whether your design is usable.

Usability Goals

Learnability

Visibility

Efficiency

Error handling

Simplicity

IAP 2010

6.470 IAP Web Programming Competition

First, let's look at some design guidelines. **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. This lecture is largely based on **Jakob Nielsen's 10 heuristics**, but another good list is **Tog's First Principles** (see the references on the last slide).

We'll classify the design principles into the usability **goals** that we're trying to satisfy:

Learnability concerns whether the interface is easy for people to learn and remember.

Visibility is about whether the interface gives feedback and makes its state easy for the user to see and understand. Software is, by default, completely invisible, so a user interface has to make extra effort to show what's going on.

Efficiency is about whether the interface is fast to operate.

Error handling is about reducing the frequency or cost of errors that the user makes.

Subjective satisfaction is about making users happier and more satisfied with the interface.

Simplicity is an overarching goal that tends to improve usability in general. Simpler interfaces – with fewer parts to understand and use – tend to be more learnable, more efficient, have fewer ways to make errors, and often more satisfying. Simplifying is often the most effective way to improve usability.

Learnability



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

Let's start with learnability. This is what many people are thinking about when they use words like "intuitive" or "user-friendly". This example that we saw at the beginning of lecture had serious problems with learnability, because it used the scrollbar in a way that's unfamiliar, inconsistent, and frankly inappropriate

Learnability by Metaphor



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

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

IAP 2010

6.470 IAP Web Programming Competition

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?

Lack of Discoverability



Source: Interface Hall of Shame

mouse over



IAP 2010

6.470 IAP Web Programming Competition

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.

People Don't Learn Instantly



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

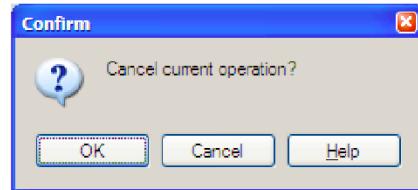
When you're designing for learnability, you have to be aware of how people actually learn. You can't assume that if the interface tells the user something, that the user will immediately learn and remember it.

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. **Just because you've said it, doesn't mean they know it.** (Incidentally, 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.)

Design Principles for Learnability

Consistency

- Similar things look and act similar
- Different things look and act different
- Consistency of wording, location, argument order, ...
- Internal consistency: within your UI
External consistency: with other UIs



Match the real world

- Use common words, not tech jargon



Source: Interface Hall of Shame

Recognition, not recall

- Labeled buttons are better than command languages
- Combo boxes are better than text boxes

IAP 2010

6.470 IAP Web Programming Competition

In designing for learnability, the most important rule 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. One 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.

There are three kinds of consistency you need to think about: **internal consistency** within your application, so that things the user learns in one place can be carried over to other places; **external consistency** with other applications on the same platform; and **metaphorical consistency** with your interface metaphor or similar real-world objects.

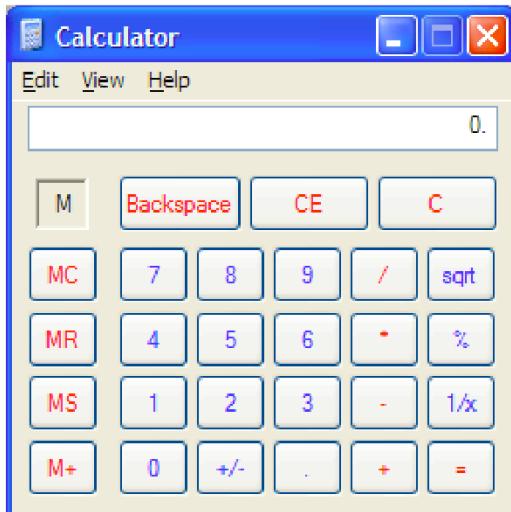
The system should **match the real world** of the user's experience as much as possible. Another way to say this is "speak the user's language." If the user speaks English, then the interface should also speak English, not Geekish. Technical jargon should be avoided. How might a user interpret the second dialog box shown above? 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.

Recognition is better than recall – i.e., if the user can operate your interface by recognizing the function they want, rather than having to recall it from memory, then their learning burden is significantly reduced. 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 GUI interfaces driven by buttons and menus rely on knowledge in the world.

Visibility



IAP 2010

6.470 IAP Web Programming Competition

We now turn to visibility – making the program's state visible to the user.

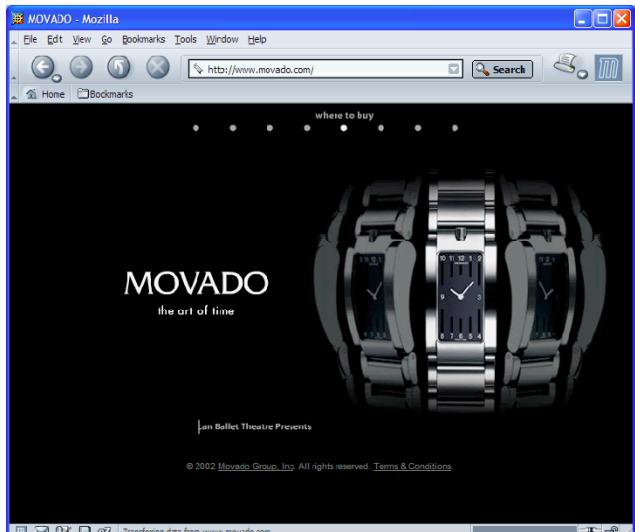
This is the Windows XP calculator. It looks and works just like a familiar desk calculator, a stable and widely-copied interface that many people are familiar with. It's a familiar metaphor, and trivial for calculator users to pick up and use. Unfortunately it deviates from the metaphor in some small ways, largely because the buttons are limited to text labels. The square root button is labeled "sqrt" rather than the root symbol. The multiplication operator is * instead of X.

But this interface adheres to its metaphor so carefully that it passes up some tremendous opportunities to improve on the desk calculator interface. Why only one line of display? A history, analogous to the paper tape printed by some desk calculators, would cost almost nothing. Why only one memory slot? Why display "M" instead of the actual number stored in memory? All these issues violate **visibility**. A more serious violation of the same heuristic: the interface actually has invisible modes. When I'm entering a number, pressing a digit appends it to the number. But after I press an operator button, the next digit I press starts a new number. There's no visible feedback about what low-level mode I'm in. Nor can I tell, once it's time to push the = button, what computation will actually be made.

(Incidentally, although this interface has good metaphorical consistency, so it's easy to learn for people who've used a pocket calculator before, it's not easily learnable if you haven't. Most of the buttons are cryptically worded, violating the principle "recognition, not recall". MC, MR, MS, and M+? What's the difference between CE and C? My first guess was that CE meant "Clear Error" (for divide-by-zero errors and the like); some people in class suggested that it means "Clear Everything". In fact, it means "Clear Entry", which just deletes the last number you entered without erasing the previous part of the computation. "C" actually clears everything.)

It turns out that this interface also lets you type numbers on the keyboard, but the interface doesn't give a hint about that possibility. In fact, in a study of experienced GUI users who were given an onscreen calculator like this one to use, 13 of 24 never realized that they could use the keyboard instead of the mouse (Nielsen, Usability Engineering, p. 61-62). One possible solution to this problem would be to make the display look more like a text field, with a blinking cursor in it, implying "type here". Text field appearance would also help the Edit menu, which offers Copy and Paste commands without any obvious selection (external consistency).

“Mystery Navigation”



Suggested by Adam Champy

IAP 2010

6.470 IAP Web Programming Competition

This is the home page for Movado, a company that makes expensive, stylish watches. The little white dots at the top of the window are menu options. If you watched the opening animation that precedes this screen, you'd see each menu label appear briefly over each dot. But if you skipped over the intro, you wouldn't see that, and you may not even realize that a menu is hiding up there under those stylish white dots.

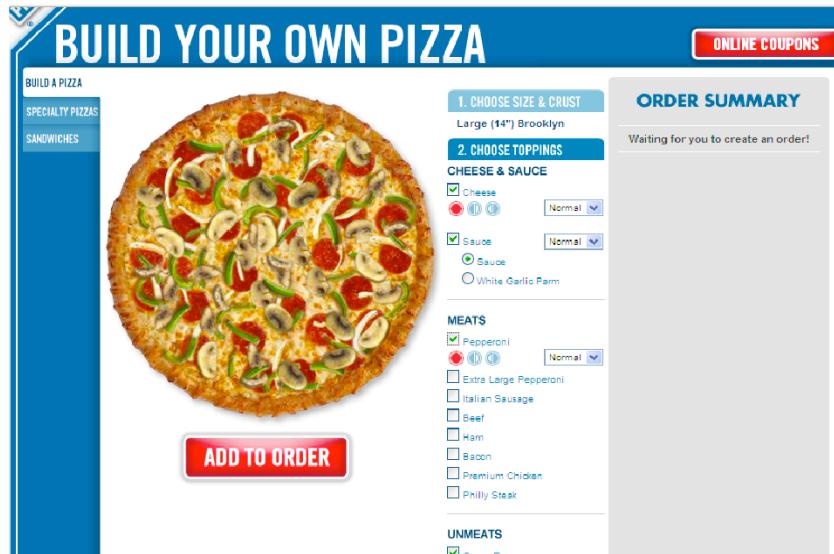
When you mouse over a dot, you actually have to wait for a cute little animation (a watch hand sweeping around the dot) before the menu label appears. Each little animation takes 2 seconds. So scanning the entire menu to look at all the options takes 16 seconds!

Clearly this is even worse than MOMA's approach, since it starts with an invisible menu interface and makes it **inefficient** to boot. More tellingly, MOMA only cares about your eyeballs, but Movado actually wants to sell you a watch. If you can't figure out their menu, or lose patience with it, you may be headed elsewhere.

One lesson you might draw from these examples is that Flash animation is bad, but that's too simplistic. Flash is a powerful tool that can be used for good or ill.

A better lesson might be that aesthetic appeal does not automatically confer usability. Effective graphic design is an important element of usability, but it isn't the whole story by any means.

Great Visibility & Feedback



IAP 2010

6.470 IAP Web Programming Competition

Here's the Domino's Pizza build-your-own-pizza process. (You can try it yourself by going to the Domino's website and clicking Order to start an order; you'll have to fill in an address to get to the part we care about, the pizza-building UI.)

Design Principles for Visibility

Make system state visible

- keep the user informed about what's going on
- Mouse cursor, selection highlight, status bar

Give prompt feedback

- Response time rules-of-thumb
 - < 0.1 sec seems instantaneous
 - 0.1-1 sec user notices, but no feedback needed
 - 1-5 sec display busy cursor
 - > 1-5 sec display progress bar

IAP 2010

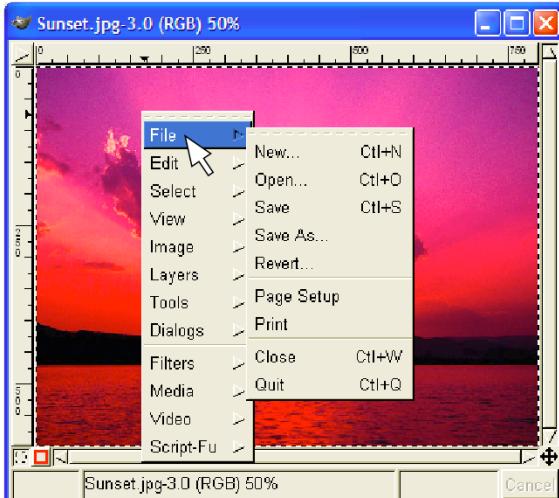
6.470 IAP Web Programming Competition

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.

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.

Efficiency



IAP 2010

6.470 IAP Web Programming Competition

Efficiency concerns how quickly an expert user can operate the system – submitting input or commands, and perceiving and processing the system's output. Note that this is typically not about the performance of the program's algorithms at all – instead, it's about the performance of the I/O channel between the user and the program. A user interface that requires fewer keystrokes to do a task is more efficient. The problem of efficiency is more subtle than just counting keystrokes, however.

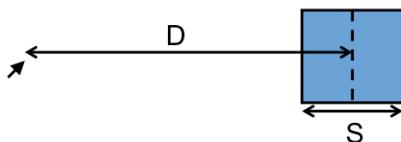
Recall the example of Gimp from the start of this lecture. All Gimp menus are accessed from a *context menu*, which pops up on right-click. You don't have to move your mouse up to the menu bar – a context menu can be popped up anywhere. So it should be faster. Right?

Wrong. With Gimp's design, as soon as the mouse hovers over a choice on the context menu (like File or Edit), the submenu immediately pops up to the right. That means, if I want to reach an option on the File menu, I have to move my mouse carefully to the right, staying within the File choice, until it reaches the File submenu. If my mouse ever strays into the Edit item, the File menu I'm aiming for vanishes, replaced by the Edit menu. So if I want to select File/Quit, I can't just drag my mouse in a straight line from File to Quit – I have to drive into the File menu, turn 90 degrees and then drive down to Quit! **Cascading submenus** are actually slower to use than a menu bar. Gimp's designers made a choice without fully considering how it interacted with human capabilities.

Pointing Tasks: Fitts's Law

How long does it take to reach a target?

- Moving mouse to target on screen
- Moving finger to key on keyboard
- Moving hand between keyboard and mouse



$$T = RT + MT = a + b \log(D/S)$$

- $\log(D/S)$ is the *index of difficulty* of the pointing task

IAP 2010

6.470 IAP Web Programming Competition

Let's look at some facts about the human motor processing system, because this will help us understand just how bad the cascading submenu problem is.

In simplified form, the human cognitive system is a feedback loop: your senses feed stimuli into your brain, your brain processes those stimuli, and your brain instructs your muscles to do something. Then your senses perceive the effect your muscles had on the world, and your brain can adjust what the muscles are doing to correct for errors. We rely on this feedback loop to walk, catch a ball, draw a straight line, put food in our mouths, and do almost everything we do.

Let's consider a common motor task in user interfaces: pointing at a target of a certain size at a certain distance away (within arm's length). The time it takes to do this task is governed by a relationship called Fitts's Law. It's a fundamental law of the human sensory-motor system, which has been replicated by numerous studies. Fitts's Law applies equally well to using a mouse to point at a target on a screen, putting your finger on a keyboard key, or moving your hand between keyboard and mouse.

We can derive Fitts's Law from a simple model of the human motor system. In each cycle, your motor system instructs your hand to move the entire remaining distance D . The accuracy of that motion is proportional to the distance moved, so your hand gets within some error ϵD of the target (possibly undershooting, possibly overshooting). Your eyes perceive where your hand arrived and compare it to the target, and then your motor system issues a correction to move the remaining distance ϵD – which it does, but again with proportional error, so your hand is now within $\epsilon^2 D$ of the target. This process repeats, with the error decreasing geometrically, until n iterations of the feedback loop have brought your hand within the target – i.e., $\epsilon^n D \leq S$. Solving for the number of cycles n , and assuming the total time T is proportional to n , we get:

$$T = a + b \log(D/S)$$

for some constants a and b .

Path Steering Tasks

Pointing vs. steering

- Fitts's Law applies only if path to target is **unconstrained**
- But the task is much harder if path is constrained to a tunnel



$$T = a + b(D/S)$$

Relationship is linear, not logarithmic

- The index of difficulty is D/S for steering, but $\log(D/S)$ for pointing
- This is why cascading menus are slow!

IAP 2010

6.470 IAP Web Programming Competition

We can also see why cascading submenus like Gimp's are hard to use, because of the correction cycles the user is forced to spend getting the mouse pointer carefully over into the submenu. Because the user must keep the mouse inside the menu tunnel, they must move it slowly enough so that the error of each cycle (ϵd where d is the distance moved in that cycle) is always less than S . Thus the distance of each cycle is $d \leq S/\epsilon$, and so the total number of cycles is proportional to D/S . That's a lot slower than the $\log(D/S)$ in Fitts's Law, which applies to unconstrained pointing – in fact, it's exponentially slower!

Gimp offers the worst possible behavior here, by making the submenu disappear as soon as the mouse pointer exits the tunnel. Microsoft Windows does it a little better – you have to hover over a choice for about half a second before the submenu appears, so if you veer off course briefly, you won't lose your target. But now we know a reason that this solution isn't ideal: it exceeds T_p , so it destroys perceptual fusion and our sense of causality. And you still have to make that right-angle turn to get into the menu.

Apple Macintosh does even better: when a submenu opens, there's a triangular zone, spreading from the mouse to the submenu, in which the mouse pointer can move without losing the submenu. The user can point straight to the submenu without unusual corrections, and without even noticing that there might be a problem.

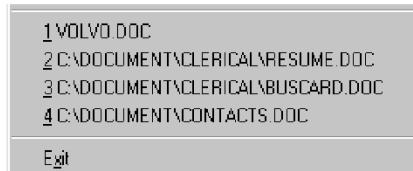
Design Principles for Efficiency

Fitts's Law and Steering Law

- Make important targets big, nearby, or at screen edges
- Avoid steering tasks

Provide shortcuts

- Keyboard accelerators
- Styles
- Bookmarks
- History



Source: Interface Hall of Shame

IAP 2010

6.470 IAP Web Programming Competition

What we've learned leads to some useful principles for making interfaces more efficient.

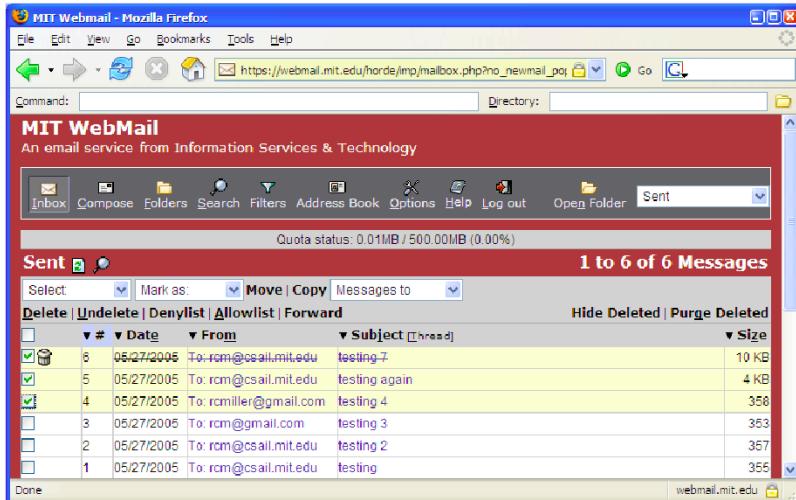
Fitts's Law has some interesting implications. The edge of the screen stops the mouse pointer, so you don't need a correcting cycle to hit it. Essentially, the edge of the screen acts like a target with infinite size. So edge-of-screen real estate is precious. The Macintosh menu bar, positioned at the top of the screen, is faster to use than a Windows menu bar (which, even when a window is maximized, is displaced by the title bar). So if you put controls at the edges of the screen, they should be active all the way to the edge to take advantage of this effect. Don't put an unclickable margin beside them.

In general, if you know that a button or control will be frequently used, then you should exploit Fitts's Law in your design of that control – make it bigger, or anchor it to the edge of the screen so that its size is effectively huge. If two buttons are frequently used together, put them close to each other. Minimize steering tasks as much as possible.

Another way to improve efficiency is by providing **shortcuts** that expert users can learn and apply to maximize their bandwidth. Keyboard combinations are a good example of this. (But be sure to strive for learnability too! If your shortcuts are too hard to learn and remember, then users won't get any benefit from them.)

Another incredibly useful kind of shortcut is a history. The recently-used files menu is a good example of this. Since users often reopen a file that they recently used, providing this history list saves them from navigating through the filesystem. Histories can be used all over an interface to save time and typing.

Errors



IAP 2010

6.470 IAP Web Programming Competition

Our fourth goal is error handling. Users make errors; you have to anticipate them, prevent them as much as possible, and deal with them well when they do happen.

Here's an example of a tricky kind of error created by the keyboard shortcuts in Mozilla Firefox and MIT Webmail. The Alt-D shortcut does different things depending on the state you're in:

- if you're browsing any other web site 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.

It's easy to see how a user habituated to expect a certain behavior from Alt-D can make serious errors here! If you press Alt-D thinking it will put the focus on the address bar, but it actually deletes an email message, then you've made a mode error.

(Thanks to InHan Kang for this example.)

Mode Error

Modes: states in which actions have different meanings

- Vi's insert mode vs. command mode
- Drawing palette

Avoiding mode errors

- Eliminate modes entirely
- Visibility of mode
- Spring-loaded or temporary modes
- Disjoint action sets in different modes



IAP 2010

6.470 IAP Web Programming Competition

Modes are states of the system 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.

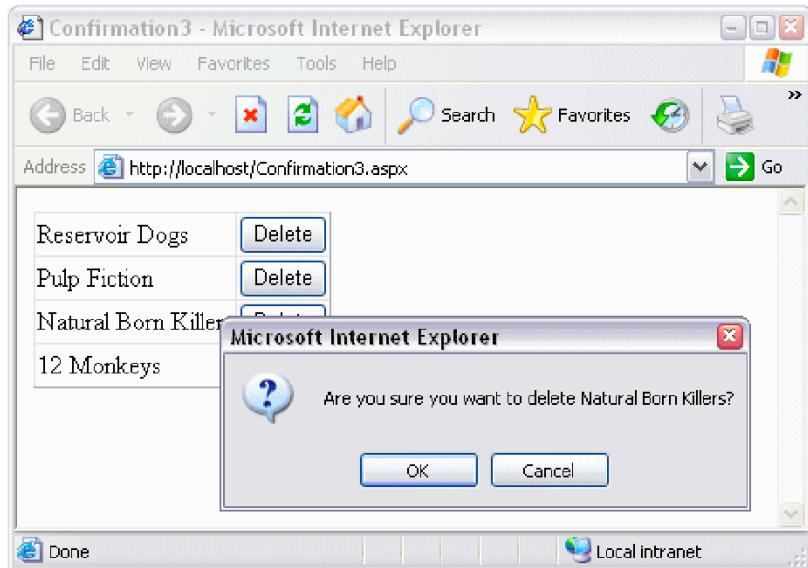
Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs.

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.

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?)

Confirmation Dialogs Aren't the Answer



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.

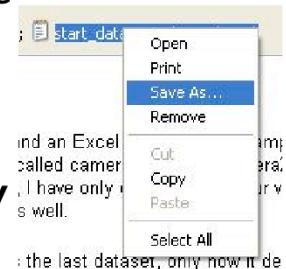
If a confirmation dialog is frequently seen – for example, every time the Delete button is pressed – then the expert users will learn to expect it, and will start to chunk it as part of the operation. In other words, to delete something, the user will learn to push Delete and then OK, without reading 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.

Design Principles for Error Handling

Prevent errors as much as possible

- Selection is better than typing
- Avoid mode errors
- Disable illegal commands
- Separate risky commands from common ones



Use confirmation dialogs sparingly

Support undo

Good error messages

- Precise
- Polite
- Constructive help



IAP 2010

6.470 IAP Web Programming Competition

One way to prevent errors is to allow users to select rather type. Misspellings then become impossible.

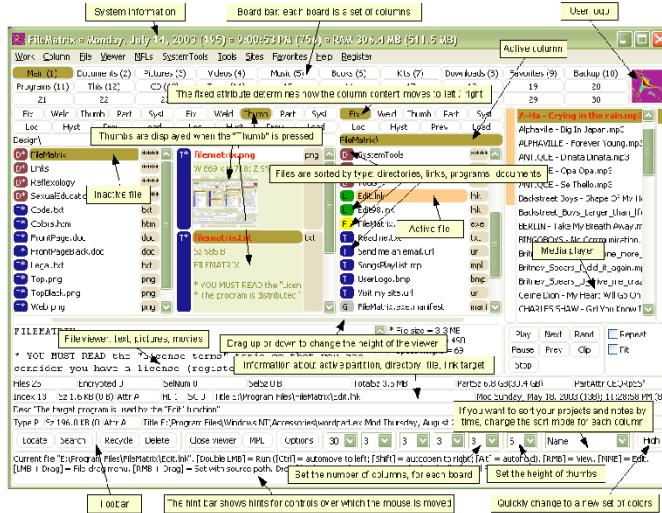
If a command is illegal in the current state of the interface – e.g., Copy is impossible if nothing is selected – then the command should be disabled (“grayed out”) so that it simply can’t be selected in the first place.

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.)

If you can’t prevent the error, at least 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.

Simplicity



Source: Alex Papadimoulis

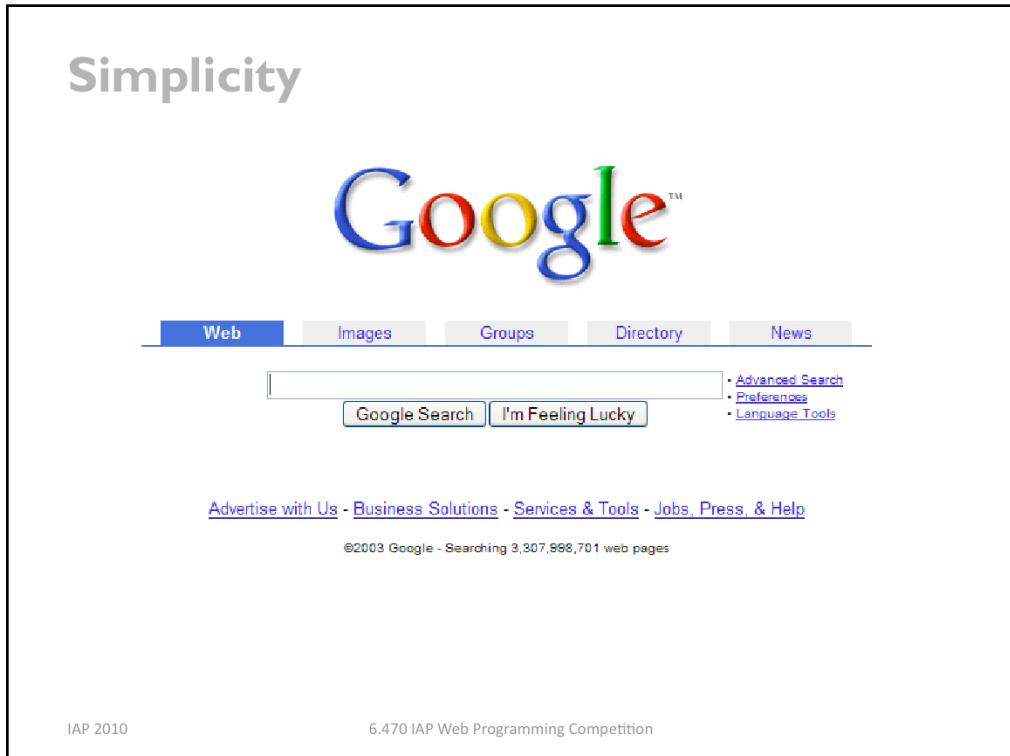
IAP 2010

6.470 IAP Web Programming Competition

The final design principle is a catch-all for a number of rules of good design, which really boil down to one word: simplicity.

This is a program called FileMatrix. I have no idea what it does, but it seems to do it all. The complexity of this interface actually interferes with a lot of our usability goals: it's less learnable (because there are so many things you have to learn), less efficient (because cramming all the functions into the window means that each button is tiny), and more error-prone (because so many things look alike).

Incidentally, this may be a good example of designing for yourself, rather than for others. The programmer who wrote this probably understands it completely, and maybe even uses a significant fraction of those features; but few other users will need that much, and it will just interfere with their ability to use it.



In contrast to the previous example, here's Google's start page. Google is an outstanding example of simplicity. Its interface is as simple as possible. Unnecessary features and hyperlinks are omitted, lots of whitespace is used. Google is fast to load and trivial to use.

Design Principles for Simplicity

“Less is More”

- Omit extraneous information, graphics, features

Good graphic design

- Few, well-chosen colors and fonts
- Group with whitespace

Use concise language

- Choose labels carefully



IAP 2010

6.470 IAP Web Programming Competition

The way to achieve simplicity is by relentless reduction of your design. 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 offers a great positive example of the less-is-more philosophy.

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.

Designing Information Displays

Title: HCI Bibliography : Human-Computer Interaction / User Interface ...

Summary: The HCI Bibliography (HCIBIB) is a free-access bibliography on Human-Computer Interaction, with over 20000 records in a searchable database. ... Learn about HCI. ...

Keywords: HCI

URL: www.hcibib.org/

Size: 14k

[HCI Bibliography : Human-Computer Interaction / User Interface ...](#)

The HCI Bibliography (HCIBIB) is a free-access bibliography on Human-Computer Interaction, with over 20000 records in a searchable database. ... Learn about HCI. ...
www.hcibib.org/ - 14k - [Cached](#) - [Similar pages](#)

[Human-Computer Interaction Resources on the Net](#)

... This is a collection of information related to Human-Computer Interaction (HCI). ...
Collections of resources for HCI researchers and practitioners. ...
www.ida.liu.se/labs/aslab/groups/um/hci/ - 9k - [Cached](#) - [Similar pages](#)

IAP 2010

6.470 IAP Web Programming Competition

Here's another example showing how redundant encoding can make an information display easier to scan and easier to use. Search engine results are basically just database records, but they aren't rendered in a simplistic caption/field display like the one shown on top. Instead, they use rich visual variables – and no field labels! – to enhance the contrast among the items. Page titles convey the most information, so they use size, hue, and value (brightness), plus a little shape (the underline). The summary is in black for good readability, and the URL and size are in green to bracket the summary.

Take a lesson from this: your program's *output displays* do not have to be arranged like *input forms*. When data is self-describing, like names and dates, let it describe itself. (This is yet another example of the **double duty** technique for achieving greater simplicity – data is acting as its own label.) And choose good visual variables to enhance the contrast of information that the user needs to see at a glance.

Design Patterns

Patterns are good solutions to common problems

➤ Breadcrumbs [Travel](#) > [Guides](#) > North America

➤ Pagination Results Page:
1 2 3 4 5 6 7 8 9 10 > [Next](#)

➤ Tabs



➤ Autocomplete

To:	yu
Cc:	Yahoo UI Blogger < yuiblogger@yahoo.com >
	Yusef Jones < yusef@somewhere.com >
Subject:	Yusef Smith < yusefs@someplace.org >

Pattern repositories

Yahoo Design Pattern Library

<http://developer.yahoo.com/ypatterns/>

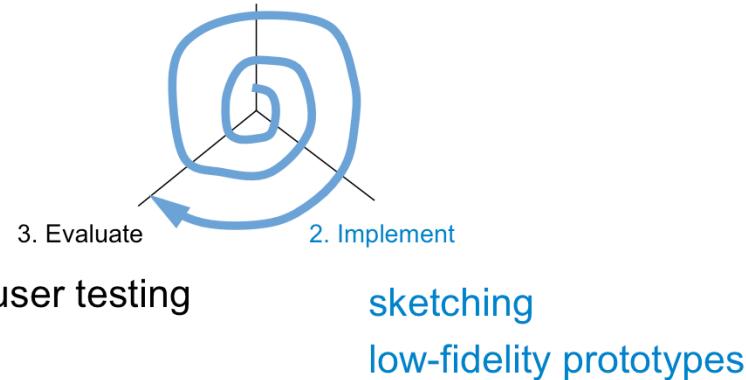
Welie.com

<http://www.welie.com/patterns/>

Lecture Outline

1. Design

design principles



IAP 2010

6.470 IAP Web Programming Competition

So we've talked about design principles, and you've developed a design. Now let's think about how to implement it cheaply.

Sketching

Paper is a very fast and effective design 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

IAP 2010

6.470 IAP Web Programming Competition

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. Not good. **Always sketch it on paper first.**

Examples of Design Sketches

Your Classes

Period	Subject	Grade
6.831	Exam: Subtotal	95
6.111	Project: Subtotal	N/A
6.470	Assignment: Subtotal	95
6.470	Midterm: Subtotal	N/A
6.111	6.111	75
6.470	6.470	80
6.470	6.470	62

This Week:
You have 6 assignments due.

Day	Assignment	Due Date	Score
Monday	Final Project	12 AM	2
Tuesday	6.111 Lab 1 Report	6 PM	3
Wednesday	6.470	5 PM	0
Thursday	6.470	1 PM	1

Add Class
+ Step 1: Basic Details
+ Step 2: Assignments

Add New Class
Subject Number: _____
Subject Name: _____
Class Website: _____
Professor Name: _____

IAP 2010

6.470 IAP Web Programming Competition

Here are some nice examples of design sketches from an earlier 6.831 class. These are alternative designs (left and right) for key pages of a grade management web site. Notice the sketchiness, the handwritten labels. But these sketches have some realistic data in them, which is a good idea, but if you find that coming up with fake data inhibits your thought process, you can often use squiggles for the data too.

Examples of Design Sketches

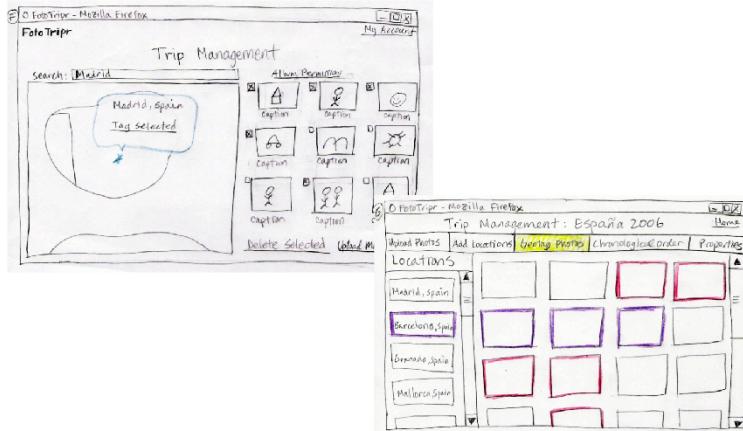
IAP 2010

6.470 IAP Web Programming Competition

Your drawings should be done by hand. For most people, hand-sketching on paper is much faster than using a drawing program. Further, hand sketches in particular are valuable because they focus attention on the issues that matter in early design, without distracting you 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 from users. 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-sketched 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.

Here’s a comparison of two early-design sketches made for the same project. The hand-sketch on the left focuses on what you care about in the early stage of design – e.g., what buttons need to be on this screen? The one on the right probably took longer to draw, and it’s full of distracting details. I have trouble looking at it without thinking, “do I really like the SWIFT SHOPPER title shown that way?” Detailed graphic design simply isn’t relevant at this stage of design, so don’t use tools that focus your attention on it.

Examples of Design Sketches



IAP 2010

6.470 IAP Web Programming Competition

It's important to brainstorm radically different design alternatives, and put them down in sketches. Here's an example from an application for plotting trip photos on a map. An important task of the problem was assigning geographical locations to photos you'd taken. These sketches show two significantly different alternatives for this task – one using a map to select a location and checkboxes to associate a subset of photos with that location, and the other using a list of locations (which can be edited on a separate screen) and color-coded highlighting to associate locations with photos.

Sketching multiple alternatives gives you the ability to talk about them with your teammates, to discuss the pros and cons, to mix and match and build on them.

Later Prototypes

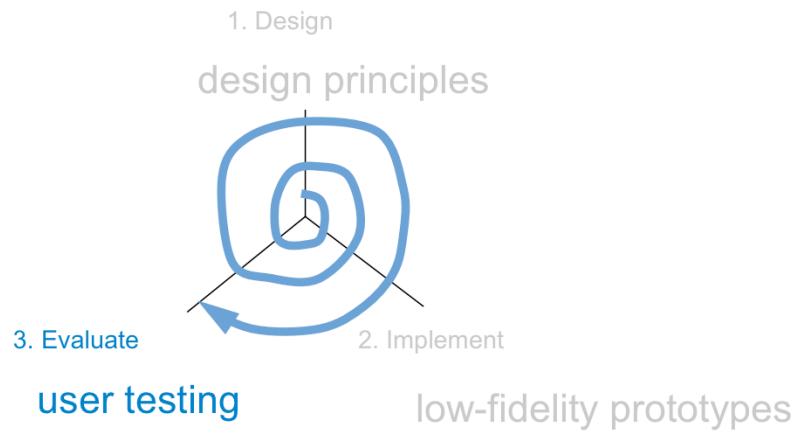
Paper sketches first!

Then static HTML + CSS wireframe

- Include some example data
- Experiment with styling and layout without generating it dynamically

Then dynamic page generated by PHP or Javascript

Lecture Outline



IAP 2010

6.470 IAP Web Programming Competition

Now that you've implemented a design as a prototype, it's time to evaluate it.

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

IAP 2010

6.470 IAP Web Programming Competition

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

IAP 2010

6.470 IAP Web Programming Competition

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 use 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 research 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 so *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.”

IAP 2010

6.470 IAP Web Programming Competition

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 and design principles in mind

Iterate over your design

Make cheap, throw-away prototypes

Evaluate them with 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>