Usability

Prof. Rob Miller
MIT EECS

Load up these web sites:
http://express.dominos.com/order/olo.jsp
http://wsdm2012.org/

User Interface Hall of Shame

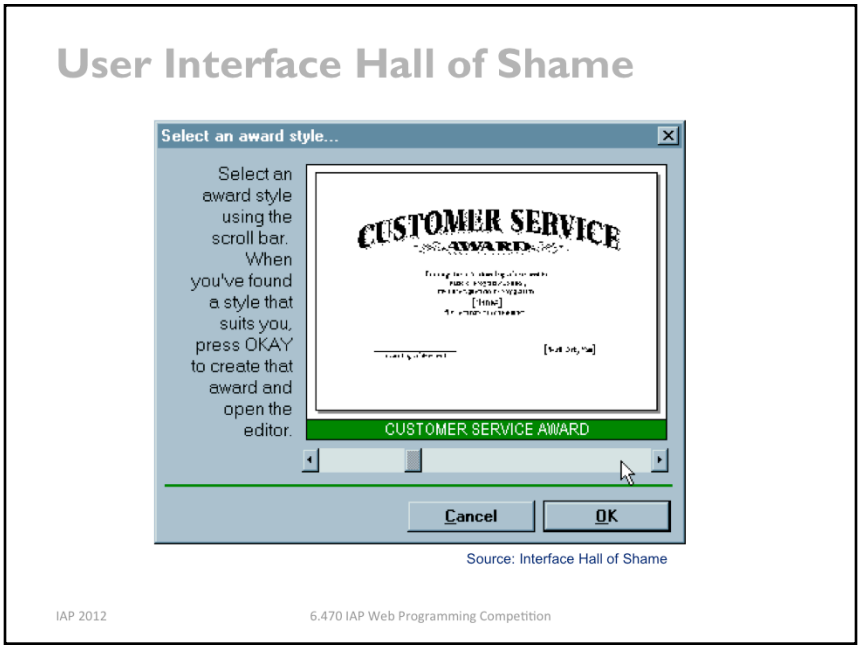
Source: Interface Hall of Shame

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.



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.

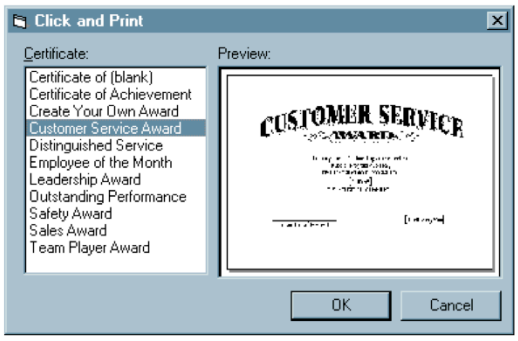
But the dialog doesn't get any better for frequent users, either. If a frequent user wants a template they've used before, how can they find it? Surely they'll remember that it's 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes 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?

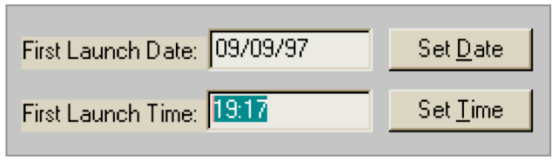
Redesigning the Interface



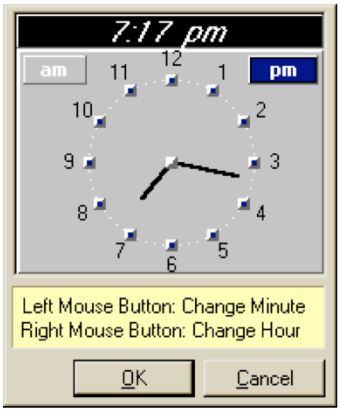
Source: Interface Hall of Shame

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 2012 6.470 IAP Web Programming Competition

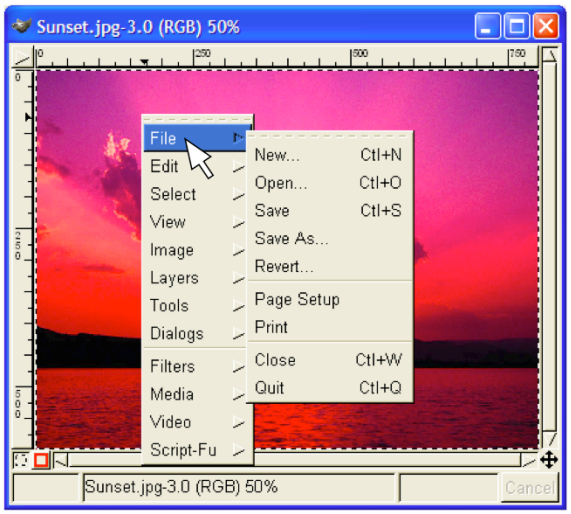
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to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



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.

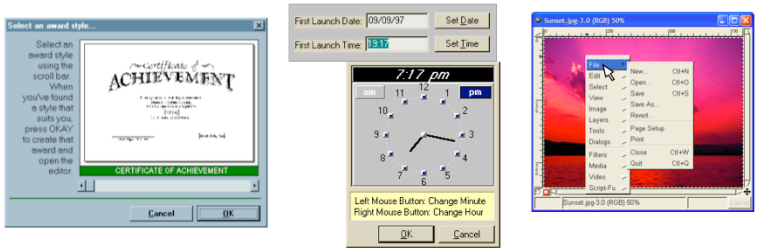
Usability Defined

Usability

- how well users can use the system's functionality

Dimensions of usability

- **Learnability**: is it easy to learn and remember?
- **Efficiency**: once learned, is it fast to use?
- **Errors**: are errors few and recoverable?



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.

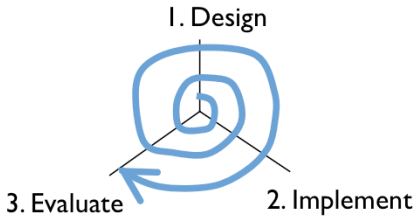
More narrowly defined, usability measures how well users can use the system's functionality. Usability has several dimensions: learnability, efficiency, and error rate/ severity (among others). Each of the examples we've looked at causes trouble for one or more of these dimensions.

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 faster to learn than interface Y, or faster to use, or causes fewer errors, for some set of tasks and some class of users.

User-Centered Design

Iterative design

- you won't get it right the first time
- plan to build a series of prototypes
- use **spiral design**
  - early stages should be low-fidelity and cheap

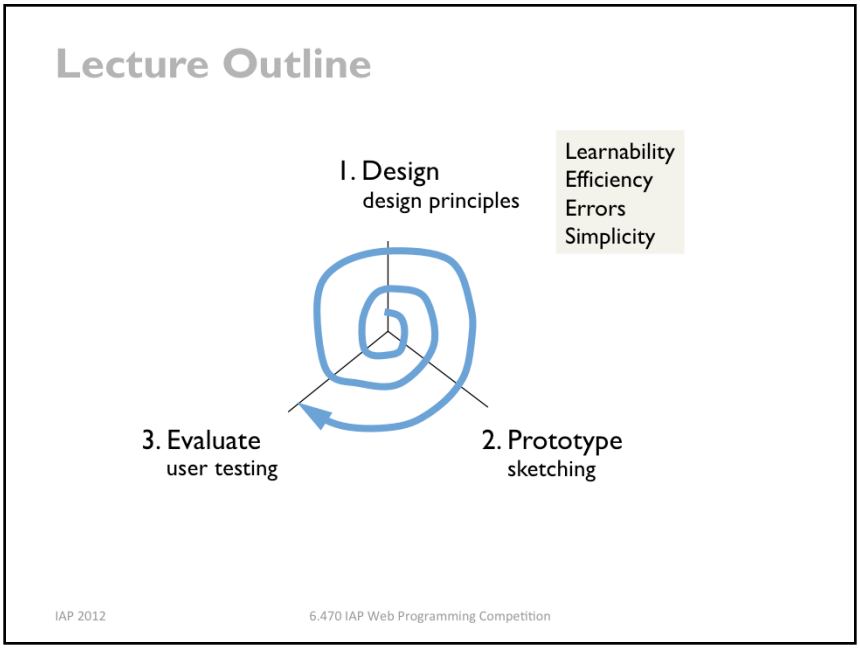


Early focus on users and tasks

- find what your users are like and what they need to do

Constant evaluation

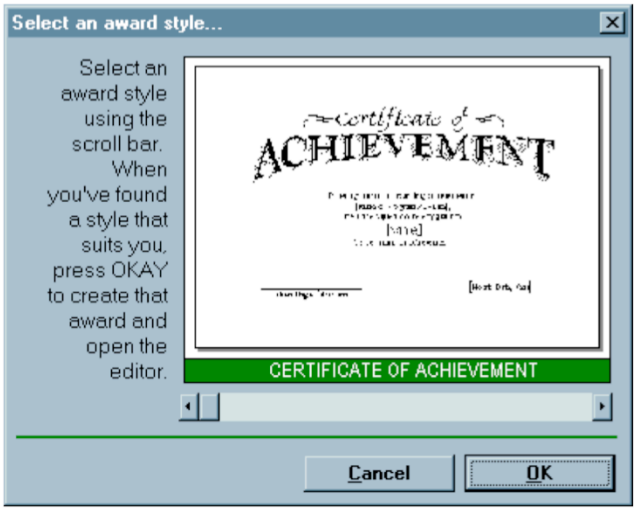
- evaluate prototypes on every iteration, using analysis, critique, and/or user testing



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.

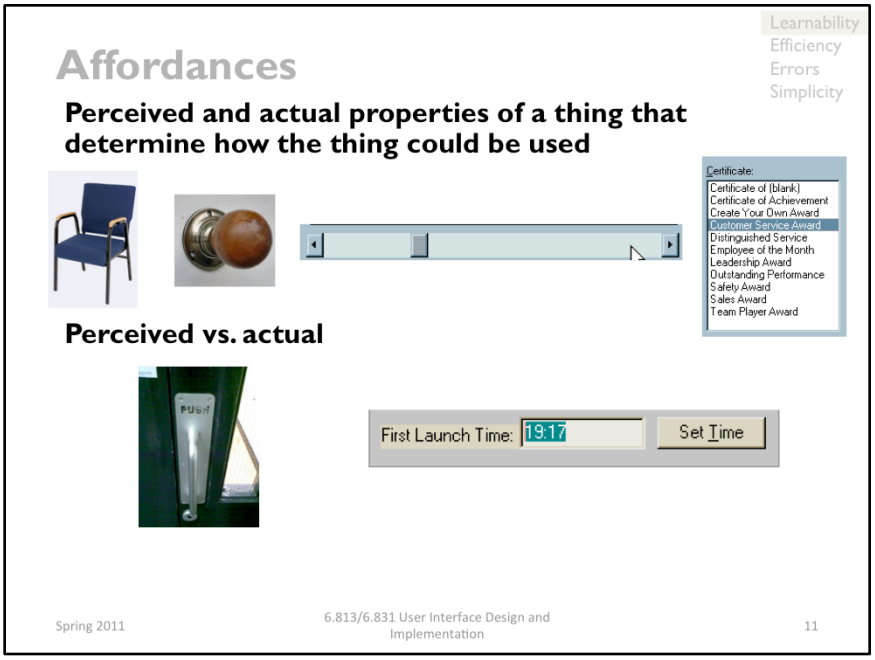
Learnability

Learnability
Efficiency
Errors
Simplicity



Source: Interface Hall of Shame

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



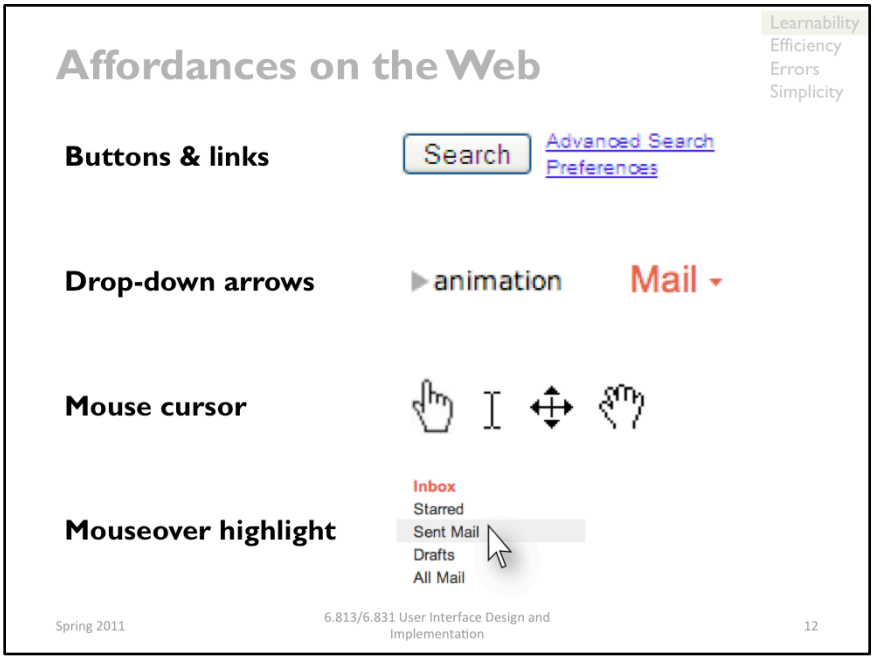
An *affordance* refers to "the perceived and actual properties of a thing", primarily the properties that determine how the thing could be operated. Chairs have properties that make them suitable for sitting; doorknobs are the right size and shape for a hand to grasp and turn. A button's properties say "push me with your finger." Scrollbars say that they continuously scroll or pan something that you can't entirely see. Affordances are how an interface communicates **nonverbally**, telling you how to operate it.

Affordances are rarely innate – they are learned from experience. We recognize properties suitable for sitting on the basis of our long experience with chairs. We recognize that listboxes allow you to make a selection because we've seen and used many listboxes, and that's what they do.

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.

Recall the textbox from one of our examples at the start, whose perceived affordance (type a time here) disagrees with what it can actually do (you can't type, you have to push the Set Time button to change it). Or the door handle shown here, whose nonverbal message (perceived affordance) clearly says "pull me" but whose label says "push" (which is presumably what it actually affords). The parts of a user interface should agree in perceived and actual affordances.

The original definition of affordance (from psychology) referred only to actual properties, but when it was imported into human computer interaction, perceived properties became important too. Actual ability without any perceivable ability is an undesirable situation. We wouldn't call that an affordance. Suppose you're in a room with completely blank walls. No sign of any exit -- it's missing all the usual cues for a door, like an upright rectangle at floor level, with a knob, and cracks around it, and hinges where it can pivot. Completely blank walls. But there *is* actually an exit, cleverly hidden so that it's seamless with the wall, and if you press at just the right spot it will pivot open. Does the room have an "affordance" for exiting? To a user interface designer, no, it doesn't, because we care about how the room communicates what should be done with it. To a psychologist (and perhaps an architect and a structural engineer), yes, it does, because the actual properties of



The first kind of visibility is for **actions**: what can the user do? (Or where can the user *go*, if we're talking about an information-rich web site?)

We've already talked about **affordances**, which are the actual and perceived properties of an object that indicate how it should be operated. Note the word *perceived* – if the user can't perceive affordances, then they won't effectively communicate. We had an example in lecture 1 of a text box that showed a selection but didn't allow editing. So it appeared to afford editing (perceived affordance), but it didn't actually afford editing (no actual affordance).

Here are some more examples of commonly-seen affordances in graphical user interfaces. Buttons and hyperlinks are the simplest form of affordance for actions. Buttons are typically metaphorical of real-world buttons, but the underlined hyperlink has become an affordance all on its own, without reference to any physical metaphor.

Downward-pointing arrows, for example, indicate that you can see more choices if you click on the arrow. The arrow actually does double-duty – it makes visible the fact that more choices are available, and it serves as a hotspot for clicking to actually make it happen.

Mouse cursor changes are another kind of affordance – a visible property of a graphical object that suggests how you operate it. When you move the mouse over a hyperlink, for example, you get a finger cursor. When you move over the corner of a window, you often get a resize cursor; when you move over a textbox, you get a text cursor (the "I-bar").

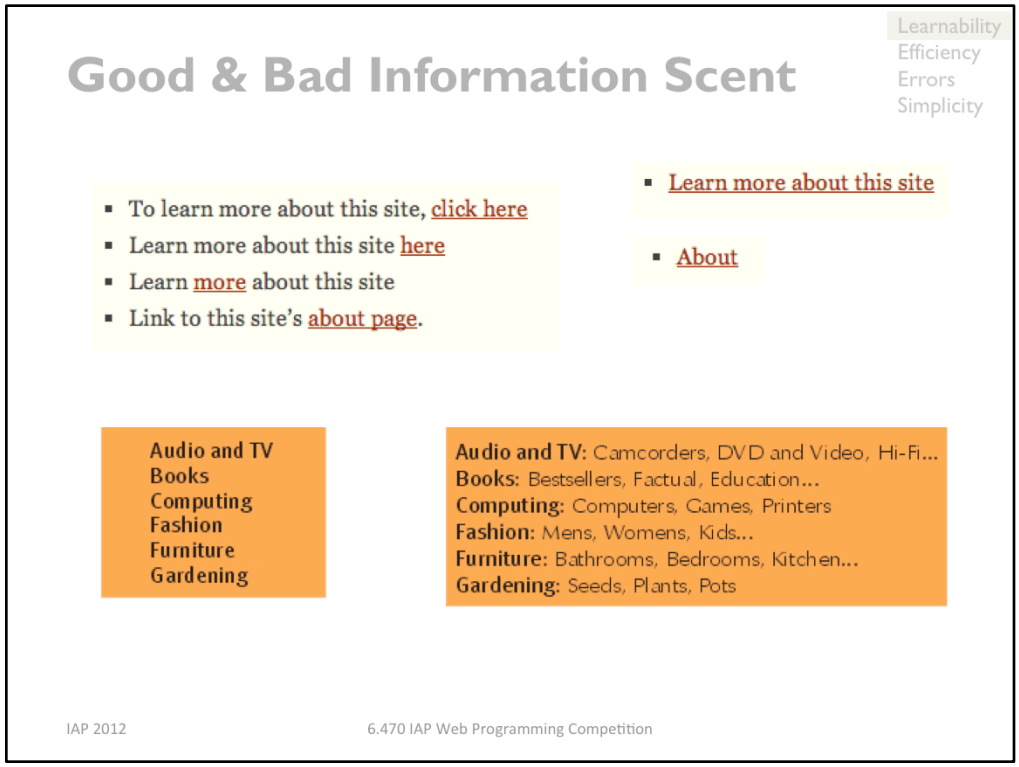
Finally, the visible highlighting that you get when you move the mouse over a menu item or a button is another kind of affordance. Because the object **visibly responds** to the presence

Hyperlinks in your interface – or in general, any kind of **navigation**, commands that go somewhere else – should provide good, appropriate information scent.

Examples of bad scent include misleading terms, incomprehensible jargon (like "Set Program Access and Defaults" on the Windows XP Start menu), too-general labels ("Tools"), and overlapping categories ("Customize" and "Options" found in old versions of Microsoft Word).

Examples of good scent can be seen in the (XP-style) Windows Control Panel on the left, which was carefully designed. Look, for example, at "Printers and Other Hardware." Why do you think printers were singled out? Presumably because task analysis (and collected data) indicated that printer configuration was a very common reason for visiting the Control Panel. Including it in the label improves the scent of that link for users looking for printers. (Look also at the icon – what does that add to the scent of Printers & Other Hardware?)

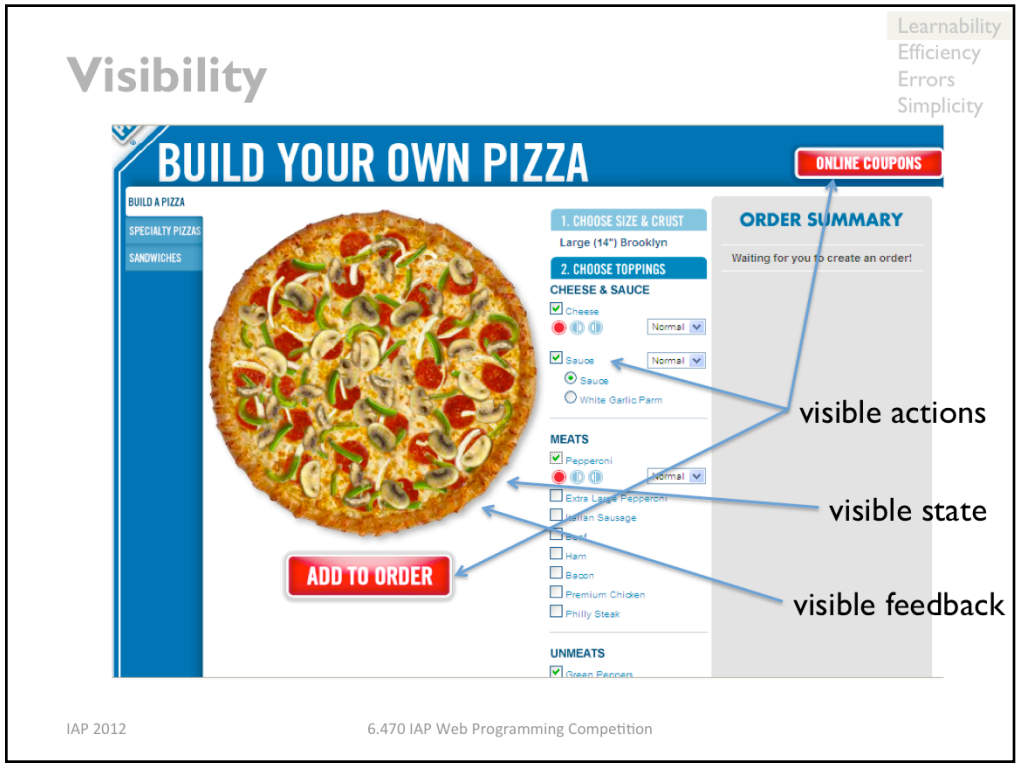
Date, Time, Language, and Regional Options is another example. It might be tempting to find a single word to describe this category – say, Localization – but its scent for a user trying to reset the time would be much worse.



Here are some examples from the web. Poor information scent is on the left; much better is on the right.

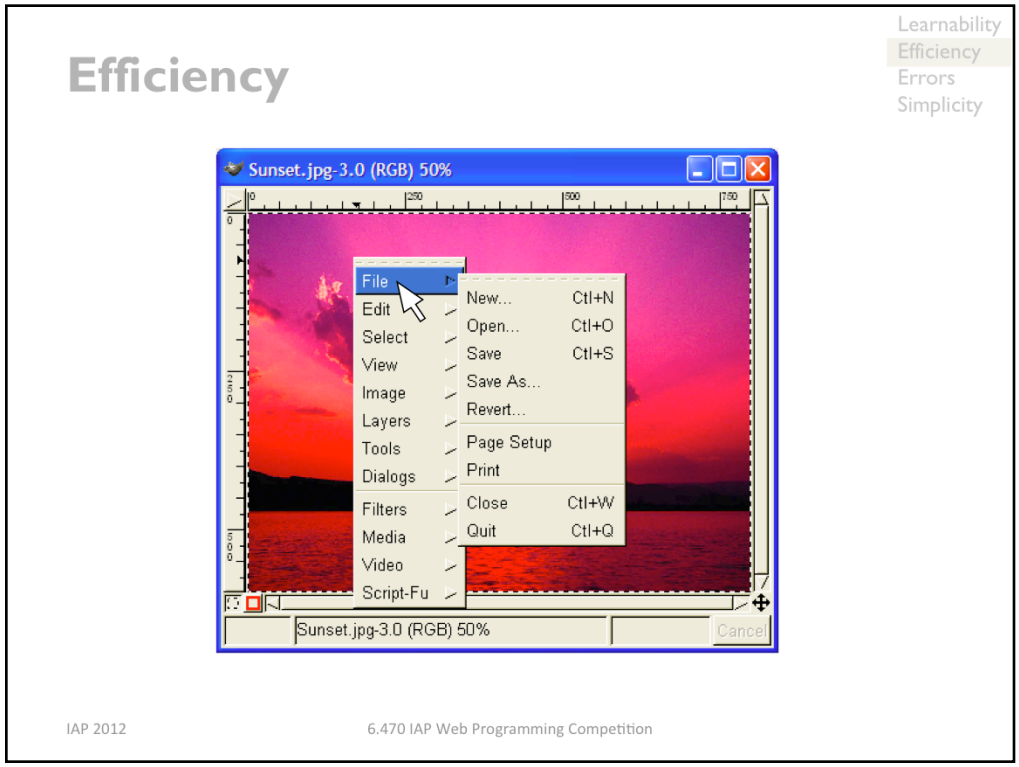
The first example shows an unfortunately common pathology in web design: the "click here" link. Hyperlinks tend to be highly visible, highly salient, easy to pick out at a glance from the web page – so they should convey specific scent about the action that the link will perform. "Click here" says nothing. Your users won't read the page, they'll scan it.

Notice that the quality of information scent depends on the user's particular goal. A design with good scent for one set of goals might fail for another set. For example, if a shopping site has categories for Music and Movies, then where would you look for a movie soundtrack? One solution to this is to put it in *both* categories, or to provide "See Also" links in each category that direct the user sideways in the hierarchy.



We've mentioned the visibility of affordances and hyperlinks. Visibility is actually a critically important design principle for usability. There are three kinds of things that you need to make visible: the actions the user can perform; the state that the interface is in; and the feedback from the user's chosen action.

As a great exemplar for visibility, 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.)



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?

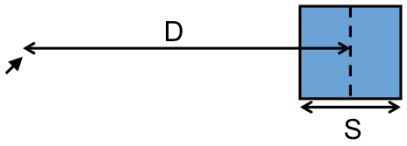
Wrong. With Gimp's design, as soon as the mouse hovers over a choice on the context menu (like File or Edit), the submenu immediately pops up to the right. That means, if I want to reach an option on the File menu, I have to move my mouse carefully to the right, staying within the File choice, until 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.

Learnability
Efficiency
Errors
Simplicity

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 2012 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 $\varepsilon^2 D$ 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., $\varepsilon^n D \leq S$. 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.

Learnability
Efficiency
Errors
Simplicity

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 2012 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<= S/ε, 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 Tp, 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.

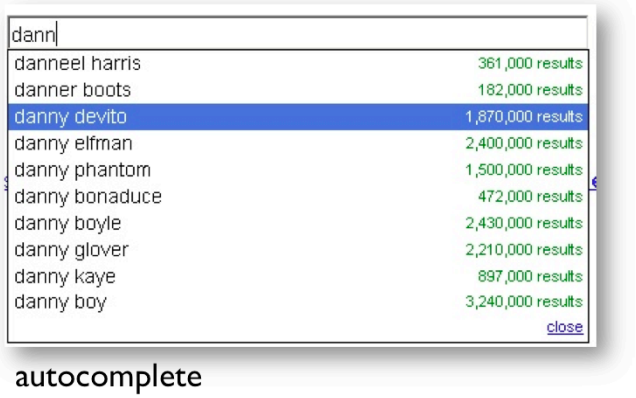
Apple Macintosh does even better: when a submenu opens, there's a triangular zone, spreading from the mouse to the submenu, in which the mouse pointer can move without losing the submenu. The user can point straight to the submenu without unusual corrections, and without even noticing that there might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

Avoid Steering Tasks

Learnability
Efficiency
Errors
Simplicity

Reduce Typing

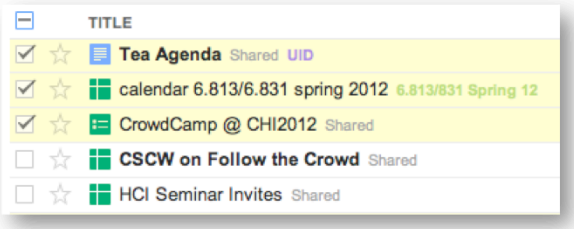
Learnability
Efficiency
Errors
Simplicity



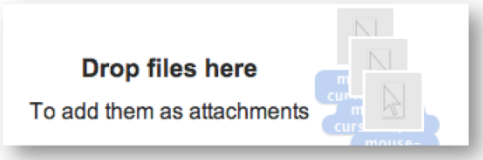
autocomplete

Aggregation

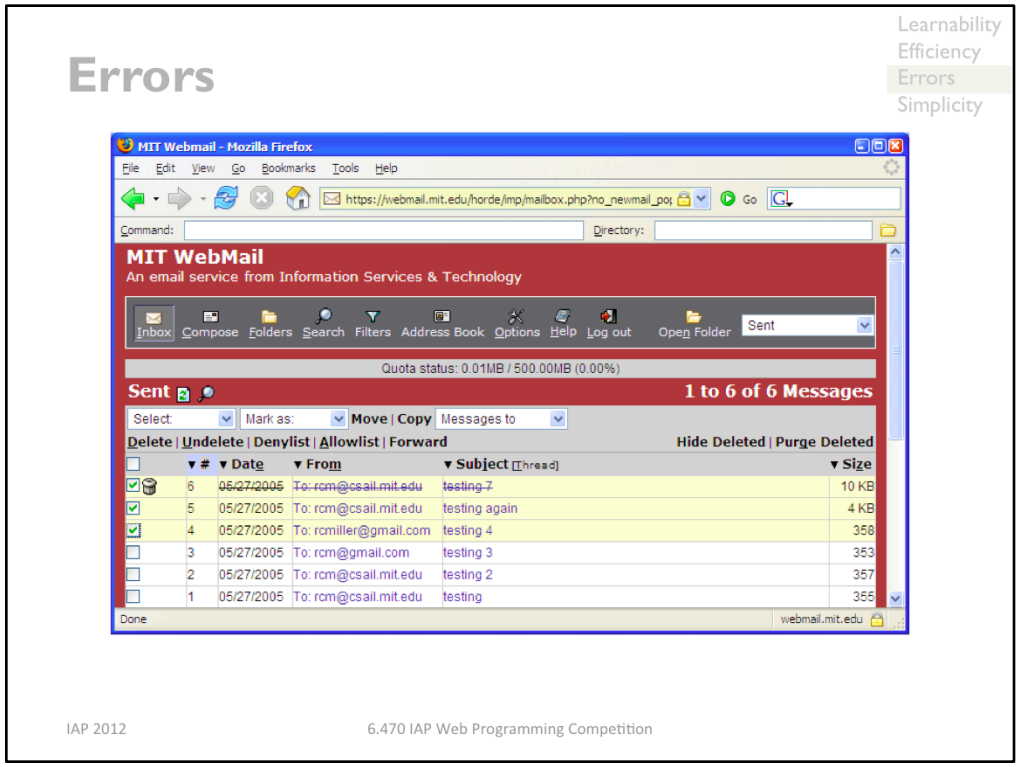
Learnability
Efficiency
Errors
Simplicity



multiple selection for action



multiple drag & drop



Our next 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.

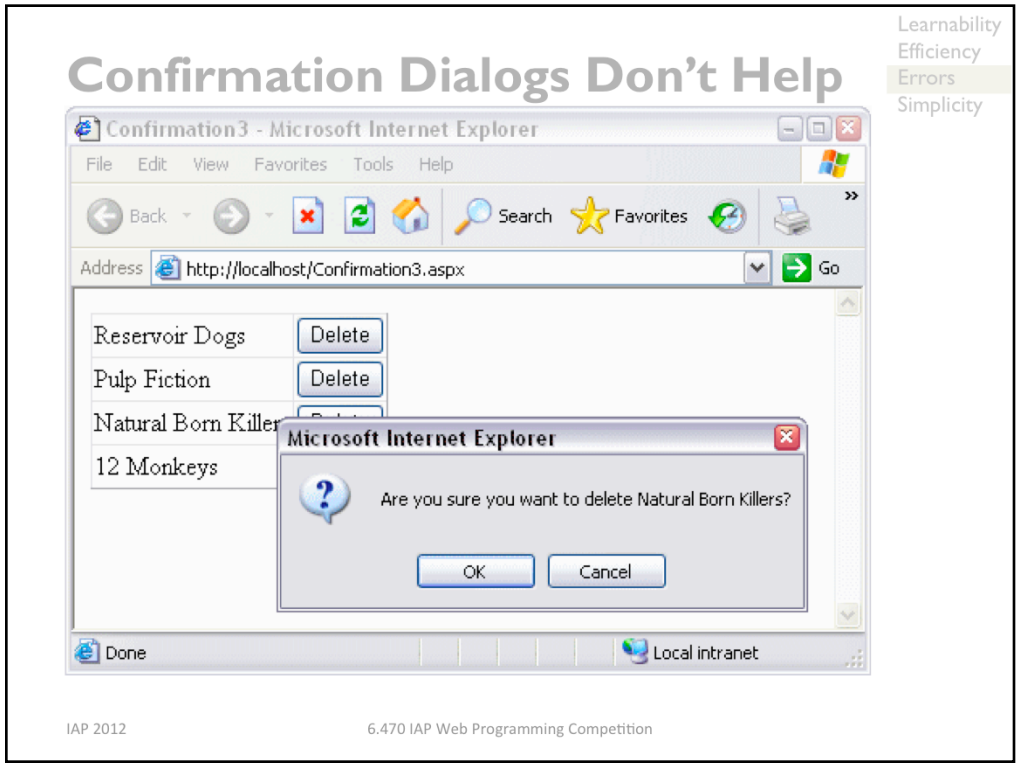
- but if you're looking at a folder in MIT Webmail, Alt-D deletes the messages you've selected.

- 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.)



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.

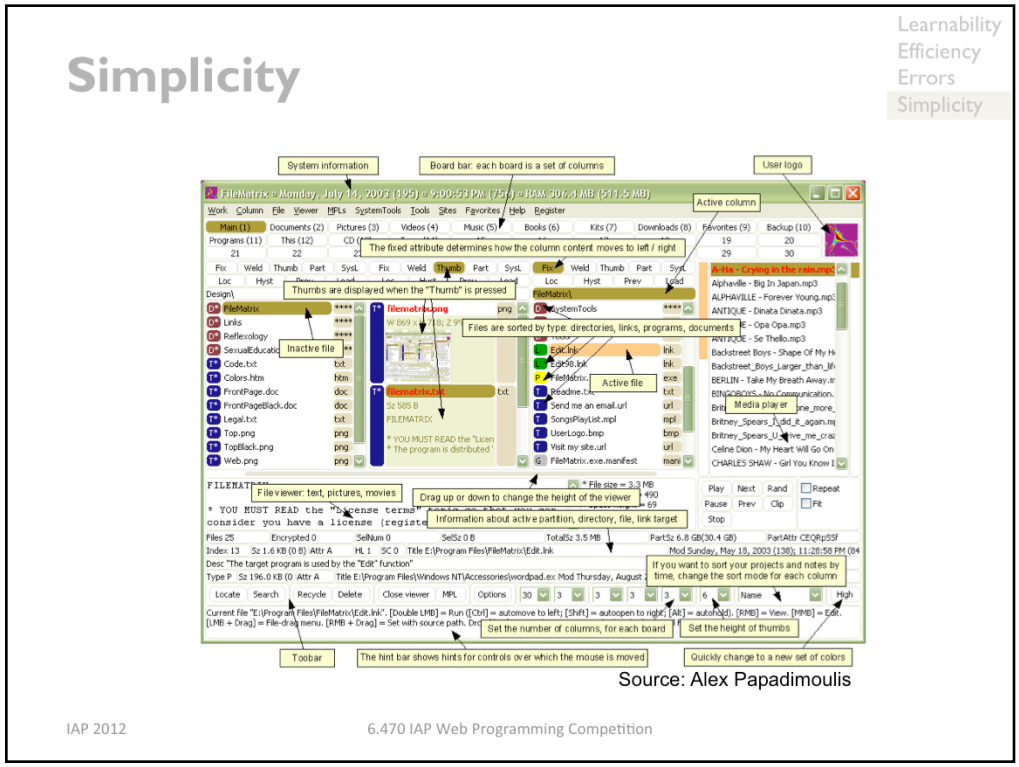
If a confirmation dialog is frequently seen – for example, every time the Delete button is pressed – then the expert users will learn to expect it, and will start to chunk it as part of the operation. In other words, to delete something, the user will learn to push Delete and then OK, without reading 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, closing the browser with unsaved edits on a form is hard to undo – but a well-designed web app could make even this undoable, using automatic save or keeping unsaved drafts in a special place.

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 a comment– 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 error correction.

Keep CRUD in mind – if you can Create an object or data field, you should be able to Read, Update, and Delete it, too.

Providing user control and freedom can have strong effects on your backend model. You'll have to make sure data are mutable. If you built your backend assuming that a user-provided piece of data would never change once it had been created, then you may have trouble building a good UI. One way that can happen is if you try to use user-provided data as a unique identifier in a database, like the user's name, or their email address, or their phone number, or the title of a document. That's generally not a good practice, because if any other object stores a reference to the identifier, then the user won't be able to edit the identifier without breaking that reference.



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).

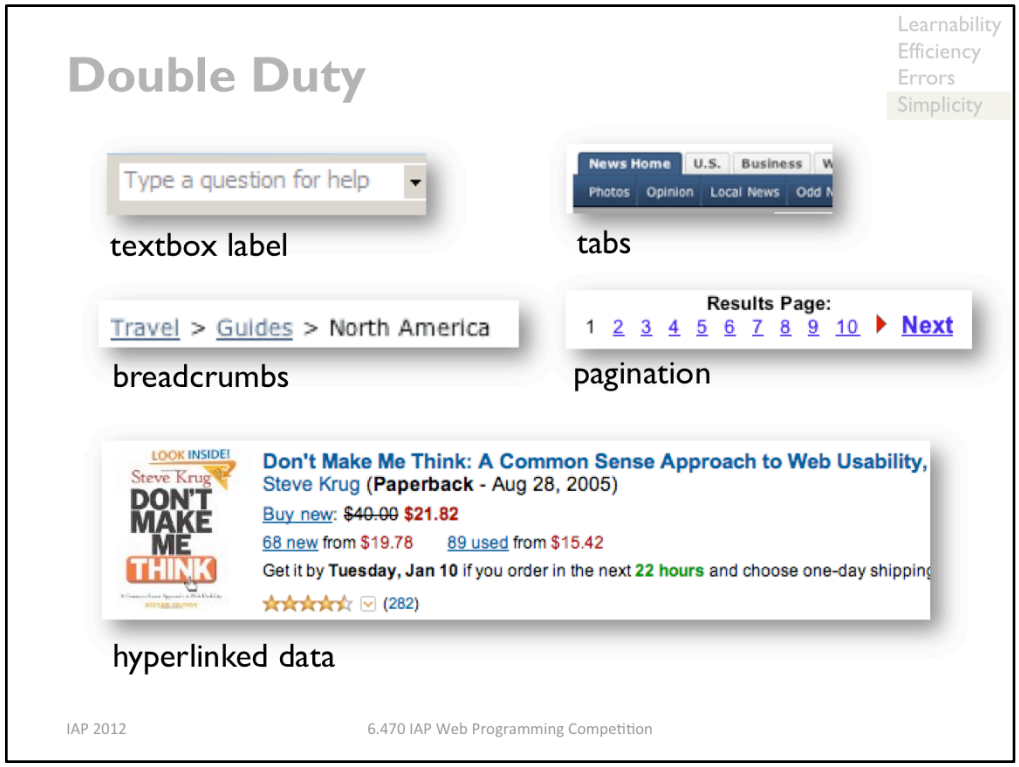
Incidentally, this may be a good example of designing for yourself, rather than for others. The programmer who wrote this probably understands it completely, and maybe even uses a significant fraction of those features; but few other users will need that much, and it will just interfere with their ability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

Reduction means that you eliminate whatever isn't necessary. This technique has three steps: (1) decide what essentially needs to be conveyed by the design; (2) critically examine every element (feature, label, UI widget, etc.) to decide whether it serves an essential purpose; (3) remove it if it isn't essential. Even if it seems essential, try removing it anyway, to see if the design falls apart.

We've already discussed the minimalism of Google. The Tivo remote is another notable example, because it's so much simpler than comparable remote controls (which tend to be dense arrays of tiny rectangular buttons, all alike). Tivo's designers aggressively removed functions from the remote, to keep it as simple as possible ("Now Preening on the Coffee Table", *New York Times*, Feb 19, 2004, http://query.nytimes.com/gst/fullpage.html?res=9c0de2d6123df93aa25751c0a9629c8b63).

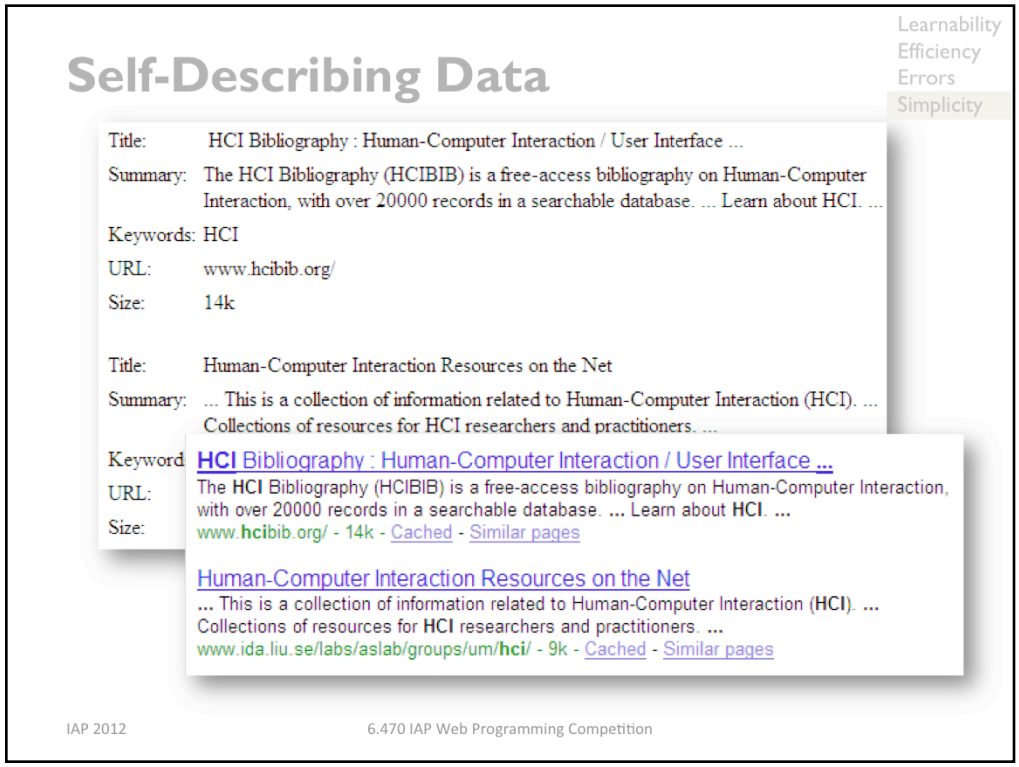
Simplicity certainly makes an interface easier to learn, because there's less to learn. But it can also make an interface faster to use and less error prone, even for an expert with that interface. Consider a remote control with 20 buttons on it. Suppose you're an expert, so you know which button to push. But you only use 3 of those buttons regularly. Which would be faster -- an interface with 3 fat buttons, or an interface with 20 little buttons? Which would be less error prone?



Another technique for simplicity is to **combine elements**, making them serve multiple roles in the design. Desktop and web interfaces have a number of patterns in which elements have multiple duties. For example, the "thumb" in a scroll bar actually serves three roles. It affords dragging, indicates the position of the scroll window relative to the entire document, and indicates the fraction of the document displayed in the scroll window. Similarly, a window's title bar plays several roles: label, dragging handle, window activation indicator, and location for window control buttons. In the classic Mac interface, in fact, even the activation indicator played two roles. When the window was activated, closely spaced horizontal lines filled the title bar, giving it a perceived affordance for dragging.

The tabs, breadcrumbs pattern and the pagination pattern also do double duty, not only showing you where you are but also providing an affordance for navigating somewhere else. Pagination links, like a scrollbar, may also show you how many pages there are.

Hyperlinking data items enables many opportunities for double-duty. Look at this snippet from Amazon: the title of the book doesn't just display its name, but it also lets the user click on it – no "more details" link needed. The image of the book cover does the same (and offers a bigger click target too, for more efficiency – but alas, it doesn't offer a visible affordance that it's clickable, so many users many not even realize it). The "68 new" and "69 used" links provide information and offer an action at the same time. The "282" at the bottom is probably the most concise example of a double-duty hyperlink, displaying the number of reviews and offering a link to get to them at the same time.



Here's another example of the **double duty** technique for achieving greater simplicity – data is acting as its own label. Search engine results are basically just database records, but they aren't rendered in a stupid caption/field display like the one shown on top. Instead, they use rich visual contrasts – and no field labels! – to distinguish among the items. The page title is the most important field, so it appears first, big, colored, with affordances for clicking (the underline). The summary is in black for good readability, and the URL and size are in green to bracket the summary.

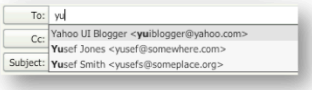
Take a lesson from this: your program's *output displays* do not have to be arranged like *input forms*. When data is self-describing, like names and dates, let it describe itself. And use good graphic design to enhance the contrast of information that the user needs to see at a glance.

Summary of Design Principles

Learnability

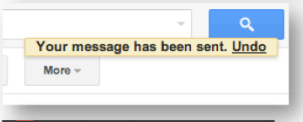
affordances
information scent
visibility of actions, state, & feedback

Efficiency



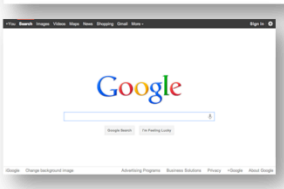
pointing vs. steering
less typing
aggregation

Errors

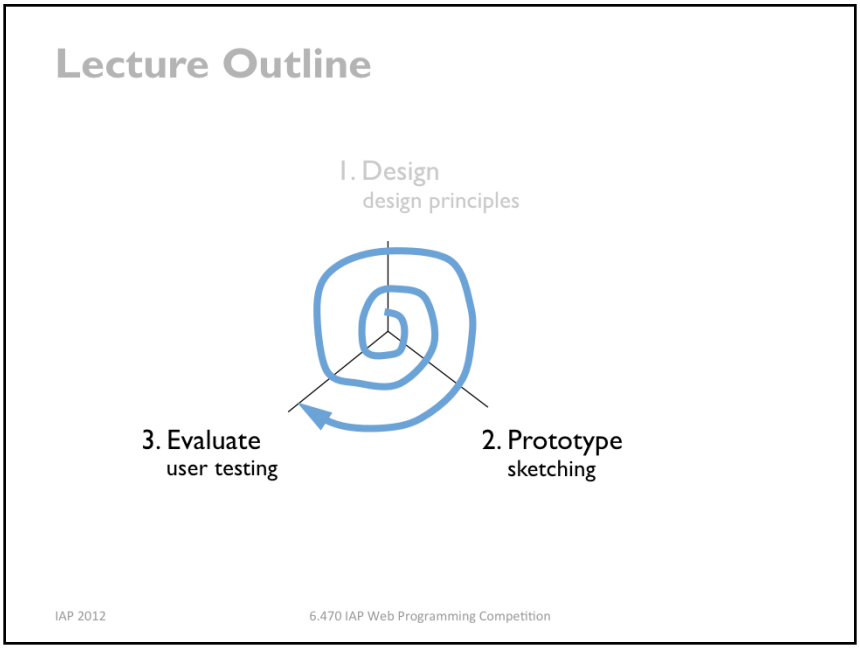


confirmation dialogs, rarely
undo
CRUD

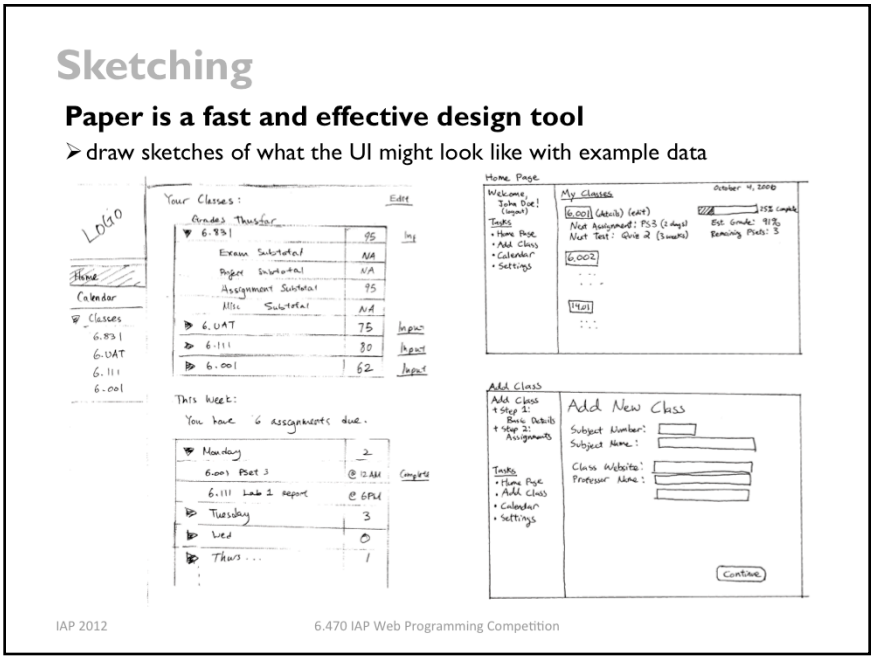
Simplicity



reduction
double-duty

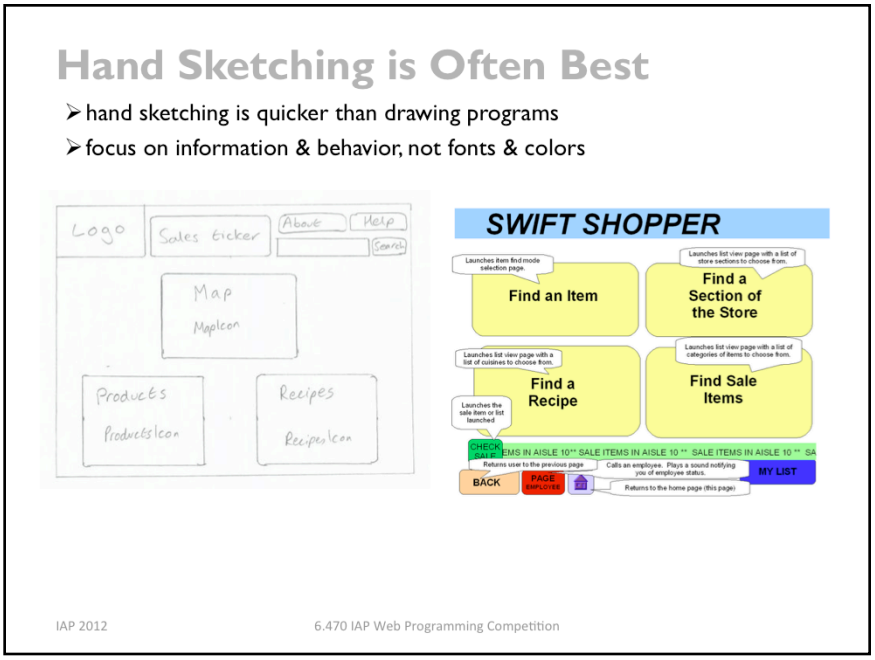


Today we're going to talk about some ideas and techniques used in user interface design: (1) **design principles** that can guide your conception of a user interface; (2) **low-fidelity prototyping** techniques that help you try out your design cheaply and easily; and (3) **user testing** to measure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



It turns out that **paper** is a terrific prototyping tool. If you sit down and write code for your UI without having drawn a sketch first, you're letting the GUI toolkit design the UI for you. Not good. **Always sketch it on paper first.**

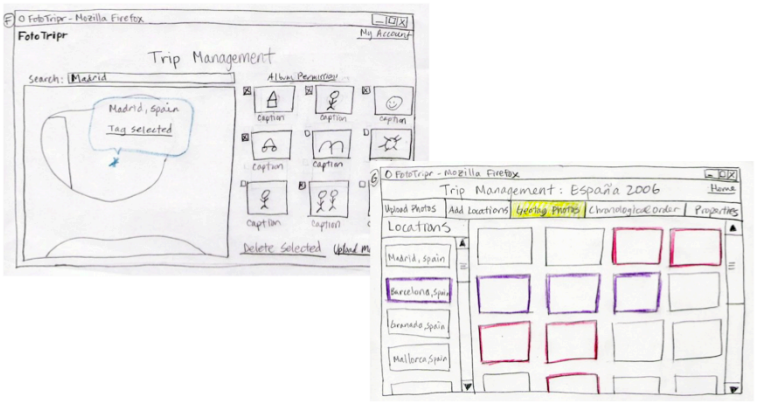
Here are some nice examples of design sketches from 6.813/6.831, the UI design class. These are alternative designs (left and right) for key pages of a grade management web site. Notice the sketchiness, the handwritten labels. But these sketches have some realistic data in them, which is a good idea, but if you find that coming up with fake data inhibits your thought process, you can often use squiggles for the data too.



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-sketch design seems less finished, less set in stone, and more open to suggestions and improvements. Architects have known about this phenomenon for many years. If they show clean CAD drawings to their clients in the early design discussions, the clients are less able to discuss needs and requirements that may require radical changes in the design. In fact, many CAD tools have an option for rendering drawings with a "sketchy" look for precisely this reason.

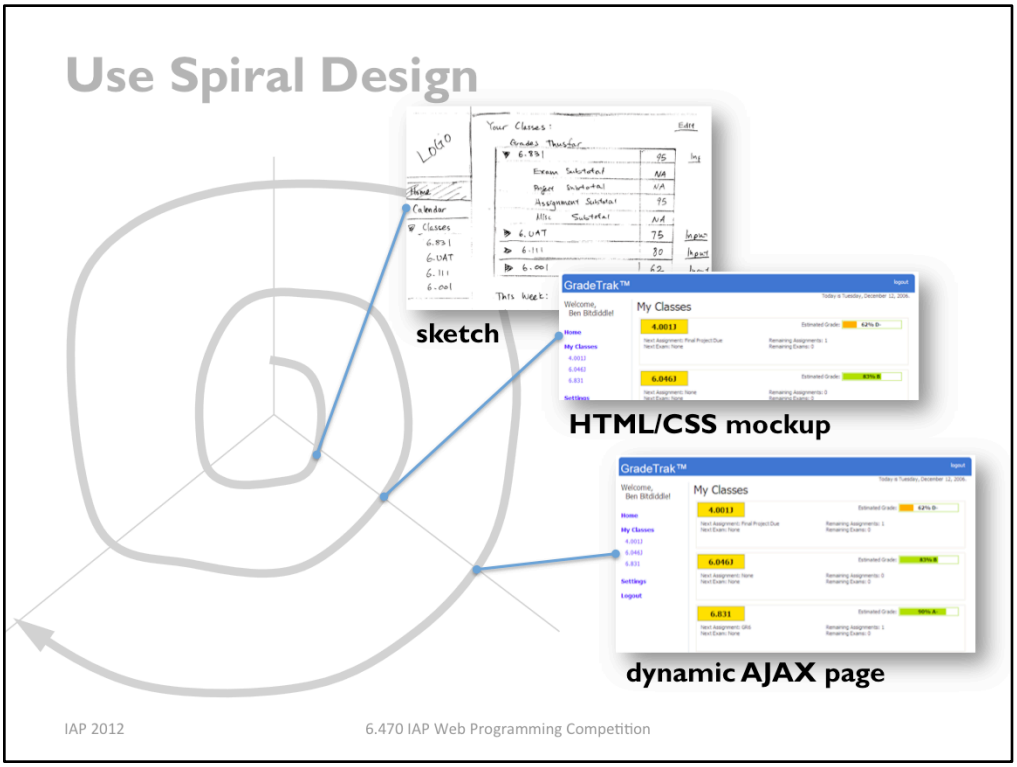
Here's a comparison of two early-design sketches made for the same project. The hand-sketch on the left focuses on what you care about in the early stage of design – e.g., what buttons need to be on this screen? The one on the right probably took longer to draw, and it's full of distracting details. 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.

Sketch Multiple Alternatives



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.

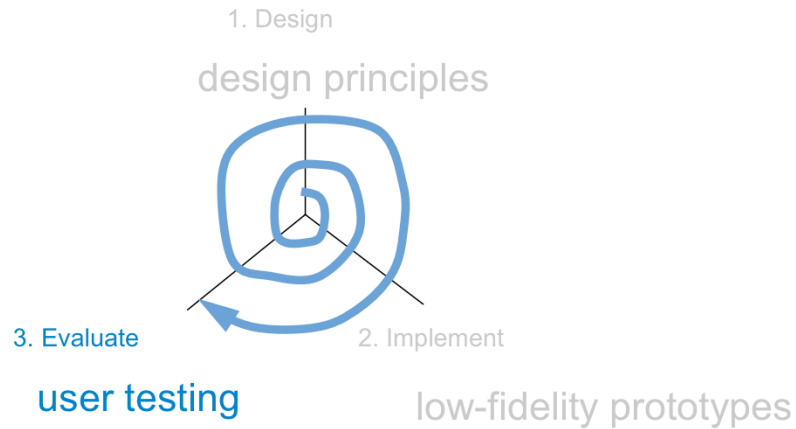


Paper sketches first!

Then a static HTML + CSS mockup. Include some example data, and experiment with styling and layout without generating it dynamically yet. If you put hyperlinks between the static wireframe pages, you can now sit somebody down and ask them to use your prototype. We'll talk about that next.

Finally, make the page dynamic, with real data from your backend.

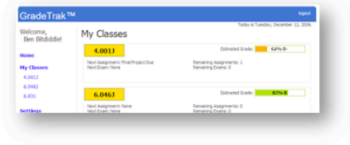
Lecture Outline



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

User Testing

Start with a prototype



Choose some representative tasks

- Make them short and concrete

"add 6.470 to your schedule"
"check your grade in 6.813"

Find some representative users

- 3-5 is often enough to find major usability problems

Watch them do tasks with the prototype

IAP 2012 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.)

icons by David Hopkins

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 2012 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 user of humans in software user studies identically with their use in psychology experiments, drug trials, and studies of new medical procedures. You have to obtain approval for a 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.

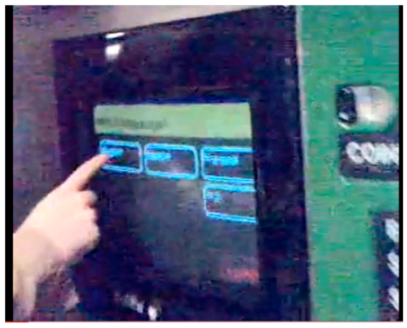
Keep yourself busy by taking a lot of notes.

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 2012 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.

Let's practice observing a user test, listening to think-aloud, and watching for critical incidents. This isn't really a user test – it's even better, it's a user interacting naturally in the wild! Watch this video of somebody using a NYC subway fare machine:

http://www.youtube.com/watch?v=mfCQbZR-nhk

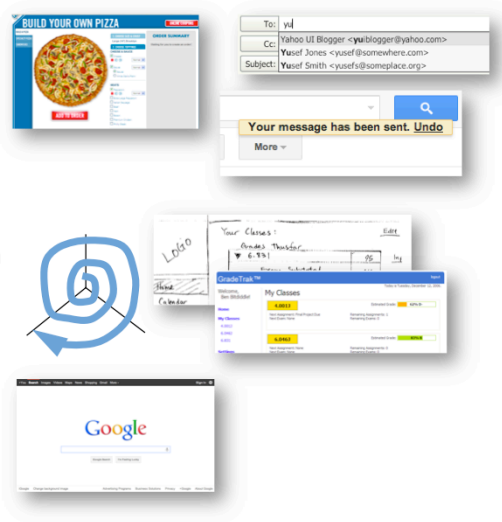
Why is the user thinking aloud? Did you note any critical incidents?

Summary

Usability
- learnability
- efficiency
- errors

User-centered design
- cheap prototypes
- evaluate constantly

Above all
- keep it simple



Further Reading

- Krug, *Don't Make Me Think*, 2005.
- Johnson, *GUI Bloopers*, 2000.
- Johnson, *Designing with the Mind in Mind*, 2010.
- Raskin, *The Humane Interface*, 2000.