L3: Learnability (continued)

- HW1 (hall of fame & shame) due Sun
- form your project group and start thinking about your project
This Flash-driven web site is the Museum of Modern Art’s Workspheres exhibition (http://www.moma.org/exhibitions/2001/workspheres/), a collection of objects related to the modern workplace. This is its main menu: an array of identical icons. Mousing over any icon makes its label appear (the yellow note shown), and clicking brings up a picture of the object.

Clearly there’s a metaphor in play here: the interface represents a wall covered with Post-it notes, and you can zoom in on any one of them.

We can praise this site for at least one reason: incredible simplicity. The designer of this site was clearly striving for aesthetic appeal. Nothing unnecessary was included. Note the use of whitespace to group the list of categories on the right, and the simple heading highlight that gives a clue to the function of the list (clicking on a category name highlights all the icons in that category).

Unfortunately, too much that was necessary was left out. Without any visible differentiation between the icons, finding something requires a lot of mouse waving. “Mystery navigation” was the term used by Vishy Venugopalan, who nominated this candidate for the UI Hall of Shame several years ago. It’s hard enough to skim the display for interesting objects to look at. But imagine trying to find an object you’ve seen before. It’s like that old card game Concentration, demanding too much recall from the user, rather than offering easy opportunities to recognize what you’re looking for.

Frankly, if real Post-it notes were arranged on a wall like this, you’d probably have just as much trouble navigating it. So the choice of metaphor may be the essence of the problem.

(Click on a few of the Post-its and note two more issues: First, how do you get back from there to the main menu? Is it internally consistent? Second, does the interface make visible which Post-its you’ve already clicked on?)
Today’s Topics

- Conceptual models of UIs
- Consistency
- Affordances
- Feedback
- Information scent
CONCEPTUAL MODELS
Regardless of interaction style, learning a new system requires the user to build a mental model of how the system works. Learnability can be strongly affected by difficulties in building that model.

A **model** of a system is a way of describing how the system works. A model specifies what the parts of the system are, and how those parts interact to make the system do what it’s supposed to do.

For example, at a high level, the model of Twitter is that there are other **users** in the system, you have a list of people that you **follow** and a list of people that follow you, and each user generates a stream of **tweets** that are seen by their followers, mixed together into a **feed**. These are all the parts of the system. At a more detailed level, tweets and people have attributes and data, and there are actions that you can do in the system (viewing tweets, creating tweets, following or unfollowing, etc.). These data items and actions are also parts of the model.
There are actually several models you have to worry about in UI design:

• The **system model** (sometimes called implementation model) is how the system actually works.

• The **interface model** (or manifest model) is the model that the system presents to the user through its user interface.

• The **user model** (or conceptual model) is how the user thinks the system works.

Note that we’re using *model* in a more general and abstract sense here than when we talk about the model-view-controller pattern. In MVC, the model is a software component (like a class or group of classes) that stores application data and implements the application behavior behind an interface. Here, a model is an abstracted description of how a system works. The *system model* on this slide might describe the way an MVC model class behaves (for example, storing text as a list of lines). The *interface model* might describe the way an MVC view class presents that system model (e.g., allowing end-of-lines to be “deleted” just as if they were characters). Finally, the *user model* isn’t software at all; it’s all in the user’s mind.
Here’s an example drawn directly from graphical user interfaces: the Back button in a web browser. What is the model for the behavior of Back? Specifically: how does the user think it behaves (the mental model), and how does it actually behave (the system model)?

The system model is that Back goes back to the last page the user was viewing, in a temporal history sequence. But on a web site that has pages in some kind of linear sequence of their own -- such as the result pages of a search engine (shown here) or multiple pages of a news article -- then the user’s mental model might easily confuse these two sequences, thinking that Back will go to the previous page in the web site’s sequence. In other words, that Back is the same as Previous! (The fact that the “back” and “previous” are close synonyms, and that the arrow icons are almost identical, strongly encourages this belief.)

Most of the time, this erroneous mental model of Back will behave just the same as the true system model. But it will deviate if the user mixes the Previous link with the Back button – after pressing Previous, the Back button will behave like Next!

Consider image editing software. Programs like Photoshop and Gimp use a pixel editing model, in which an image is represented by an array of pixels (plus a stack of layers). Programs like PowerPoint and Illustrator, on the other hand, use a structured graphics model, in which an image is represented by a collection of graphical objects, like lines, rectangles, circles, and text. In this case, the choice of model strongly constrains the kinds of operations available to a user. You can easily tweak individual pixels in Microsoft Paint, but you can’t easily move an object once you’ve drawn it into the picture.
Similarly, most modern text editors model a text file as a single string, in which line endings are just like other characters. But it doesn’t have to be this way. Some editors represent a text file as a list of lines instead. When this implementation model is exposed in the user interface, as in old Unix text editors like `ed`, line endings can’t be deleted in the same way as other characters. `ed` has a special join command for deleting line endings.

text editor: one-dimensional sequence of characters; cursor is an insertion point

typewriter: two-dimensional page; cursor is a rectangle on the page

different effects of space, return, backspace
But a wrong user model can lead to problems, as well. Consider a household thermostat, which controls the temperature of a room. If the room is too cold, what’s the fastest way to bring it up to the desired temperature? Some people would say the room will heat faster if the thermostat is turned all the way up to maximum temperature. This response is triggered by an incorrect mental model about how a thermostat works: either the timer model, in which the thermostat controls the duty cycle of the furnace, i.e. what fraction of time the furnace is running and what fraction it is off; or the valve model, in which the thermostat affects the amount of heat coming from the furnace. In fact, a thermostat is just an on-off switch at the set temperature. When the room is colder than the set temperature, the furnace runs full blast until the room warms up. A higher thermostat setting will not make the room warm up any faster. (Norman, Design of Everyday Things, 1988)

These incorrect models shouldn’t simply be dismissed as “ignorant users.” (Remember, the user is always right! If there’s a consistent problem in the interface, it’s probably the interface’s fault.) These user models for heating are perfectly correct for other systems: a car heater and a stove burner both use the valve model. And users have no problem understanding the model of a dimmer switch, which performs the analogous function for light that a thermostat does for heat. When a room needs to be brighter, the user model says to set the dimmer switch right at the desired brightness.

The problem here is that the thermostat isn’t effectively communicating its model to the user. In particular, there isn’t enough feedback about what the furnace is doing for the user to form the right model.
Try It: Design a New Thermostat

• Design a thermostat that communicates its true model (switch) effectively to a new user
  – Work with your neighbors
  – Sketch your designs
  – Come up with more than one

• Things to think about
  – Would it work to print an explanation on the thermostat? If so, what exactly would it say?
  – Think about a sink faucet: why is it easy to tell whether it’s a valve or a switch?
Here are some possible design approaches:

- a light that shows the furnace is on, to communicate that the system model’s state has only one bit (on or off). Will this work? How can it be misinterpreted?

- a display that shows the heating rate is “100%.” Plus some controls that let you apparently reduce the rate, but actually lie, because they're disconnected.

- a display that estimates the time it will take to heat to the set temperature. (See the Nest thermostat shown here, http://www.nest.com/).
CONSISTENCY
There’s a general principle of learnability: **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.
There are three kinds of consistency you need to worry about: **internal consistency** within your application; **external consistency** with other applications on the same platform; and **metaphorical consistency** with your interface metaphor or similar real-world objects.

The RealCD interface has problems with both metaphorical consistency (CD jewel cases don’t play; you don’t open them by pressing a button on the spine; and they don’t open as shown), and with external consistency (the player controls aren’t arranged horizontally as they’re usually seen; and the track list doesn’t use the same scrollbar that other applications do).
Metaphors are one way you can bring the real world into your interface. We started out by talking about RealCD, an example of an interface that uses a strong metaphor in its interface. A well-chosen, well-executed metaphor can be quite effective and appealing, but be aware that metaphors can also mislead. A computer interface must deviate from the metaphor at some point -- otherwise, why aren’t you just using the physical object instead? At those deviation points, the metaphor may do more harm than good. For example, it’s easy to say “a word processor is like a typewriter,” but you shouldn’t really use it like a typewriter. Pressing Enter every time the cursor gets close to the right margin, as a typewriter demands, would wreak havoc with the word processor’s automatic word-wrapping.

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

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

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

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

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

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

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

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

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

Other good examples of mapping include:

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

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

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

The controls on the RealCD interface don’t have a natural mapping. Why not?

Here’s a quick exercise. Consider the lights in this classroom, and design a panel of light switches to control the room’s lights, for installation next to one of the entrance doors. Devise a natural mapping between your switch panel and the lights it controls, so that a user can easily learn and remember how to use it. Don’t stop with just one design, but sketch out a few.

A few things to think about: (1) It may not make sense to control every light individually. How should the lights be grouped? (2) Think about consistency. Will your panel be recognizable as light switches from across the room? On the other hand, are there better choices than the standard North American flip switches (3) If you use flip switches, how should they be oriented?
Another important kind of consistency, often overlooked, 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. Don’t get creative when you’re writing text for a user interface; keep it simple and uniform, just like all technical writing.

Here are some examples from the Course VI Underground Guide web site – confusion about what’s a “review” and what’s an “evaluation”.

**External consistency in wording** is important too – in other words, speak the user’s language as much as possible, rather than forcing them to learn a new one. If the user speaks English, then the interface should also speak English, not Geekish. Technical jargon should be avoided. Use of jargon reflects aspects of the system model creeping up into the interface model, unnecessarily. How might a user interpret the dialog box shown here? One poor user actually read type as a verb, and dutifully typed M-I-S-M-A-T-C-H every time this dialog appeared. The user’s reaction makes perfect sense when you remember that most computer users do just that, type, all day. But most programmers wouldn’t even think of reading the message that way. Yet another example showing that you are not the user.

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

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

Incidentally, there seems to be a contradiction between these guidelines. Speaking the User’s Language argues for synonyms and aliases, so a command language should include not only delete but erase and remove too. But Consistency in Wording argued for only one command name, lest the user wonder whether these are three different commands that do different things. One way around the impasse is to look at the context in which you’re applying the heuristic. When the user is talking, the interface should make a maximum effort to understand the user, allowing synonyms and aliases. When the interface is speaking, it should be consistent, always using the same name to describe the same command or object. What if the interface is smart enough to adapt to the user – should it then favor matching its output to the user’s vocabulary (and possibly the user’s inconsistency) rather than enforcing its own consistency? Perhaps, but adaptive interfaces are still an active area of research, and not much is known.
Summary

- Conceptual models
  - system vs. interface vs. user models
- Consistency
  - internal, external, metaphorical
This message used to appear when you tried to delete the contents of your Internet Explorer cache from inside Windows Explorer (i.e., you browse to the cache directory, select a file containing one of IE’s browser cookies, and delete it).

Next time, we’ll talk about this message and others like it – not just from a learnability point of view, but also thinking about efficiency.