L2: Learnability

UI Hall of Fame or Shame?

Source: Interface Hall of Shame

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

Why is it called "Real"? Because its designers based it on a real-world object: a plastic CD case. This interface has a *metaphor*, an analogue in the real world. Metaphors are one way to make an interface more learnable, 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 obvious? Learnability comes from more than just metaphor.

UI Hall of Shame!

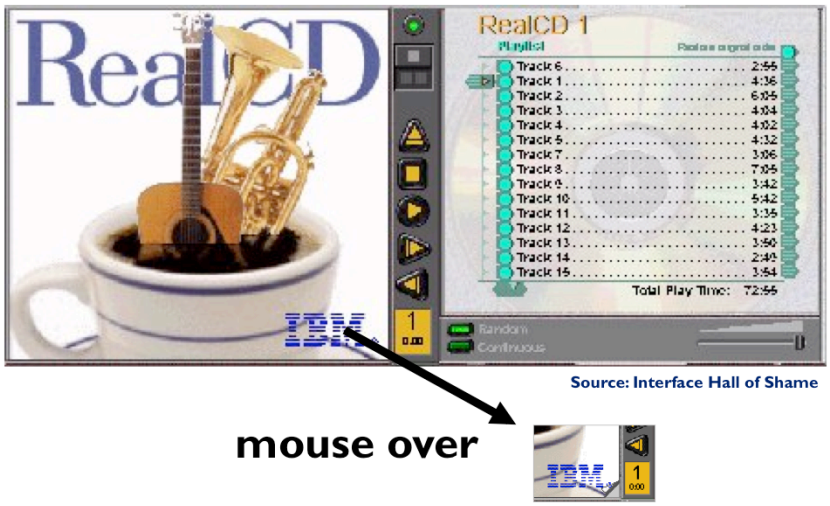
Source: Interface Hall of Shame

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

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

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

UI Hall of Shame



Source: Interface Hall of Shame

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 **help** 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 and the next, we'll look at some other ways to achieve learnability.

Topics

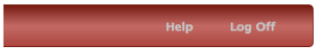
- Learning approaches
- Interaction styles
- Conceptual models of UIs

LEARNING APPROACHES

How We Learn a New User Interface

Not by reading a manual*

Not by taking a class*



Not by reading the help first*

* Standard caveat: "it depends"

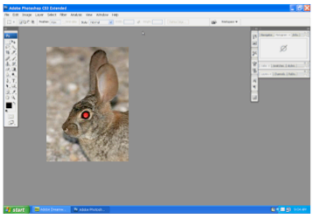
When computers first appeared in the world, there were some assumptions about how people would learn how to use the software. Programmers assumed that users would read the manual first – obviously not true. Companies assumed that their employees would take a class first – not always true. Even now that we have online help built into virtually every desktop application, and web page help often just a search engine query away, users don't go to the help first or read overviews.

All these statements have to be caveated, because in some circumstances – some applications, some tasks, some users – these might very well be the way the user learns. Very complex, professional-level tools might well be encountered in a formal training situation – that's how pilots learn how to use in-cockpit software, for example. And some users (very few of them) *do* read manuals.

Nearly all the general statements we make in this class should be interpreted as "It Depends." There will be contexts and situations in which they're not true, and that's one of the complexities of UI design.

Learning by Doing

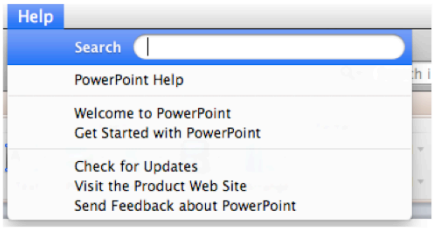
- User has a **goal** to achieve
  - "Get rid of the redeye from my photo."
- User **explores** interface for features that satisfy the goal

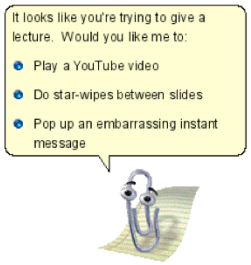


So users don't try to learn first – instead, they typically try to do what they want to do, and explore the interface to see if they can figure out how to do it. This practice is usually called **learning by doing**, and it means that the user is starting out with a goal already in mind; they are more interested in achieving that goal than in learning the user interface (so any learning that happens will be secondary); and the burden is on the user interface to clearly communicate how to use it and help the user achieve their first goal at the same time.

Seeking Help

- User resorts to seeking help when they get stuck
  - So they already have a problem when they arrive, and they're usually looking for concrete solutions to it





Only when they get stuck in their learning-by-doing will a typical user look for help. This affects the way help systems should be designed, because it means most users (even first-timers) are arriving at help with a goal already in mind and an obstacle they've encountered to achieving that goal. A help system that starts out with a long text explaining The Philosophy of the System will not work. That philosophy will be ignored, because the user will be seeking answers to their specific problem.

Modern help systems understand this, and make it easy to ask for the user to ask the question up-front, rather than wading through pages of explanation.

Lessons for Designers

- Know the user's goals when we design
- User interface should communicate how it works and how to use it
- Help must be searchable and goal-directed

The fact that users are learning our interfaces by actually **using** them has some implications for how we should design them.

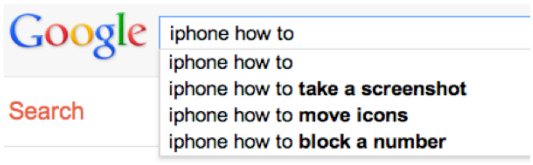
First, we should know something about what the users' goals actually are – collecting information about that is a critical feature of the user-centered design process that we'll talk about in a few lectures. If we're designing for the wrong goals, users are going to struggle to figure out how to do what they want in our system.

Second, the UI should be the **primary** teacher of how to use it. The UI itself must communicate as clearly as possible how it's supposed to be used, so that users can match their goals with appropriate actions in the system. In the next lecture, we'll talk about a few specific techniques for doing this – affordances, feedback, and information scent.

Third, when the user does have to resort to help, that help should be searchable and goal-directed. Providing a 30-minute video tutorial probably won't help people who learn by doing.

Try It:
Google Autosuggest to Find Learnability Problems

- Look at the suggested queries for prefixes like:
  - photoshop how to
  - powerpoint how to
  - iphone how to
  - android how to
- What kinds of goals do you see?
- What kinds of goals **don't** appear?
- What does it say about the learnability of the UI for that task?



Search engines have become even more important than in-application help systems, however. And a wonderful thing about search engines is that they show us query suggestions, so we can get some insight into the goals of thousands of other users. What is it that they're trying to do with their iPhone, but isn't easily learnable from the interface? (Adam Fourney, Richard Mann, and Michael Terry. "Characterizing the Usability of Interactive Applications Through Query Log Analysis." CHI 2011.)

Learning by Watching

How did you learn Alt-Tab?

One more way that we learn how to use user interfaces is by watching other people use them. That's a major way we navigate an unfamiliar subway system, for example.

Unfortunately much of our software – whether for desktops, laptops, tablets, or smartphones – is designed for one person, and you don't often use it together with other people, reducing the opportunities for learning by watching. Yet seeing somebody else do it may well be the **only** way you can learn about some features that are otherwise invisible. For example, you probably know how to use Alt-Tab to switch between windows. How did you learn that? The UI itself certainly didn't communicate it to you. Pinch-zooming on smartphones and tablets is similar – but pinch-zooming may have benefited from mass media advertising showing us all how to use it.

Social computing is changing this situation somewhat. We'll look at Twitter in a moment, and see that you can learn some things from other people even though they're not sitting next to you.

Picoquiz

Consider the new Google Glass UI. (Search the web if you haven't heard of it.) Which of the following learning approaches should its designers focus on? (**choose one**)

A. learning by reading a manual
B. learning by taking a class
C. learning by doing
D. learning by watching

To help you check your understanding and prepare for the nanoquiz in class, the lecture notes include a "picoquiz" question after every section. The picoquiz questions are similar to the nanoquiz questions you can expect to see in class, but you can check your answers yourself.

Note that the picoquiz has **all** the questions from this lecture, including questions from later in the lecture that you may not have read the material for yet. You can do each question as you get to it in your reading, and then submit the whole quiz when you've finished reading.

To answer the picoquiz questions in this lecture, go to:
http://courses.csail.mit.edu/6.831/2013/picoquiz?**lectureId=2**

INTERACTION STYLES

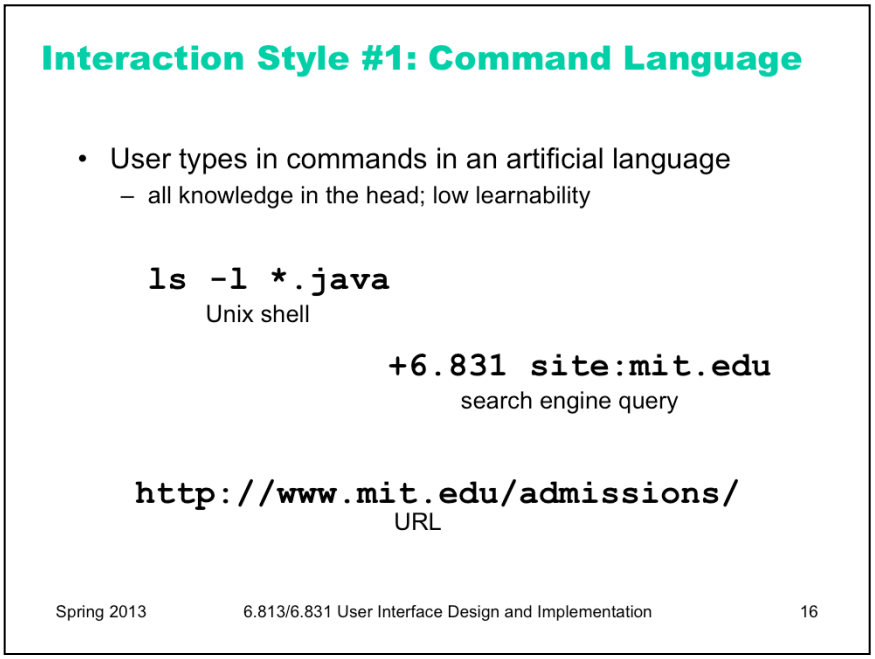
Recognition vs. Recall

- **Recognition**: remembering with the help of a visible cue
  - aka "Knowledge in the world"
- **Recall**: remembering with no help
  - aka "Knowledge in the head"
- Recognition is much easier!

It's important to make a distinction between **recognition** (remembering with the help of a visible cue, also known as **knowledge in the world**) and **recall** (remembering something with no help from the outside world – purely **knowledge in the head**). Recognition is far, far easier than uncued recall.

Psychology experiments have shown that the human memory system is almost unbelievably good at recognition. In one study, people looked at 540 words for a brief time each, then took a test in which they had to determine which of a pair of words they had seen on that 540-word list. The result? 88% accuracy on average! Similarly, in a study with 612 short sentences, people achieved 89% correct recognition on average.

Note that since these recognition studies involve so many items, they are clearly going beyond working memory, despite the absence of elaborative rehearsal. Other studies have demonstrated that by extending the interval between the viewing and the testing. In one study, people looked briefly at 2,560 pictures, and then were tested *a year* later – and they were still 63% accurate in judging which of two pictures they had seen before, significantly better than chance. One more: people were asked to study an artificial language for 15 min, then tested on it *two years later* – and their performance in the test was better than chance.

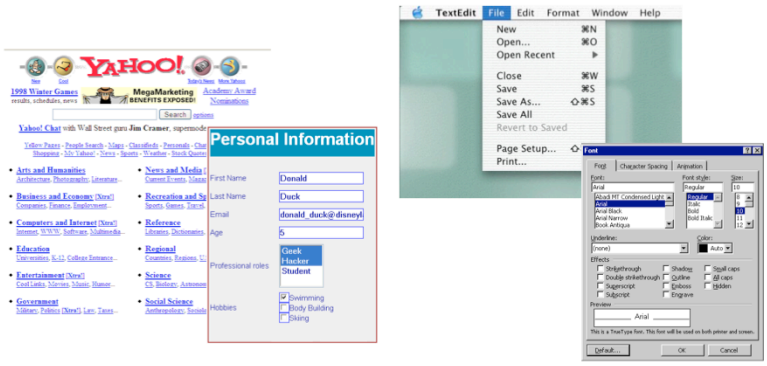


The earliest computer interfaces were command languages: job control languages for early computers, which later evolved into the Unix command line.

Although a command language is rarely the first choice of a user interface designer nowadays, they still have their place – often as an advanced feature embedded inside another interaction style. For example, Google's query operators form a command language. Even the URL in a web browser is a command language, with particular syntax and semantics.

Interaction Style #2: Menus and Forms

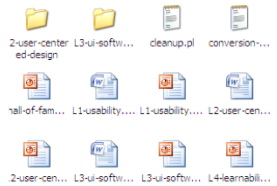
- User is prompted to choose from menus and fill in forms
  - all knowledge in the world: far more learnable



A menu/form interface presents a series of menus or forms to the user. Traditional (Web 1.0) web sites behave this way. Most graphical user interfaces have some kind of menu/forms interaction, such as a menubar (which is essentially a tree of menus) and dialog boxes (which are essentially forms).

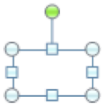
Interaction Style #3: Direct Manipulation

- User interacts with visual representation of data objects
  - Continuous visual representation
  - Physical actions or labeled button presses
  - Rapid, incremental, reversible, immediately visible effects



Files & folders on desktop

Scrollbar



Selection handles

Next we have direct manipulation: the preeminent interface style for graphical user interfaces. Direct manipulation is defined by three principles [Shneiderman, *Designing the User Interface*, 2004]:

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

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

3. The effects of actions should be **rapid** (visible as quickly as possible), **incremental** (you can drag the scrollbar thumb a little or a lot, and you see each incremental change), **reversible** (you can undo your operation – with physical actions this is usually as easy as moving your hand back to the original place, but with labeled buttons you typically need an Undo command), and **immediately visible** (the user doesn't have to do anything to see the effects; by contrast, a command like "cp a.txt b.txt" has no immediately visible effect).

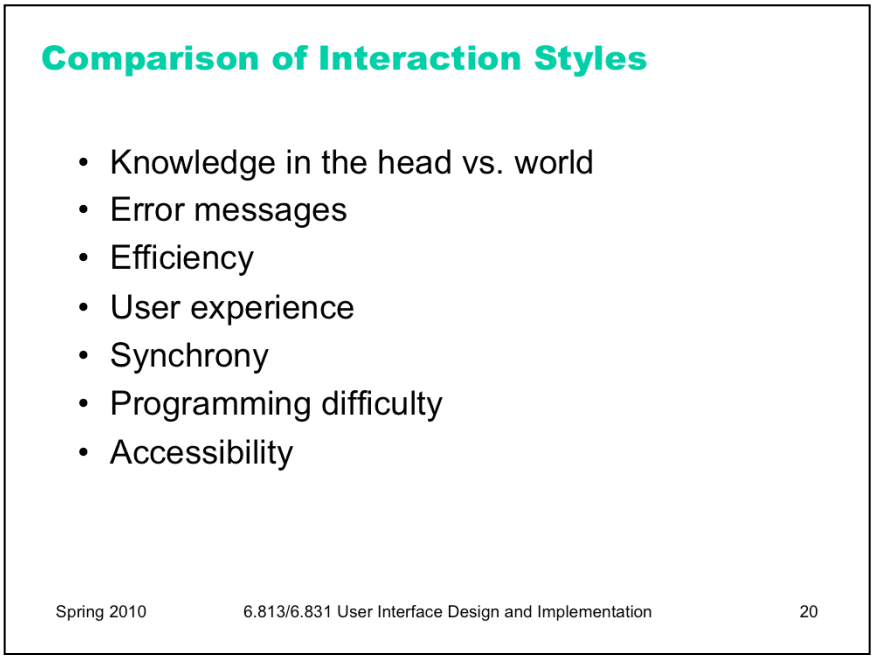
Why is direct manipulation so powerful? It exploits perceptual and motor skills of the human machine – and depends less on linguistic skills than command or menu/form interfaces. So it's more "natural" in a sense, because we learned how to manipulate the physical world long before we learned how to talk, read, and write.

Interaction Style #4: Speech Dialog

- User speaks in natural language, and system responds the same way

A fourth interaction style -- once the province of research, but now increasingly important in real deployed apps – is speech dialog in natural language. (This exchange is from the Mercury system, a flight-search system developed at MIT in the 1990s, which could be used over the phone.)

Speech dialog leans heavily on knowledge in the head. Much of this knowledge is "natural" -- in the sense that humans learn how to speak and understand their native language very early in our lives, and we have a special innate facility for spoken interaction. But beyond the mechanics of speaking, the user still needs to learn **what you can say**. What functionality is available in the system? What can I ask for? This is a fundamental problem even in human-human interaction, and is the reason why fast-food restaurant drive-through windows display a menu.



Let's compare and contrast the three styles: command language (CL), menus and forms (MF), direct manipulation (DM), and speech dialog (SD).

Learnability: knowledge in the head vs. knowledge in the world. CL requires significant learning. Users must put a lot of knowledge into their heads in order to use the language, by reading, training, practice, etc. (Or else compensate by having manuals, reference cards, or online help close at hand while using the system.) The MF style puts much more information into the world, i.e. into the interface itself. Well-designed DM also has information in the world, delivered by the affordances, feedback, and constraints of the visual metaphor. Since recognition is so much easier than recall, this means that MF and DM is much more learnable and memorable than CL or SD.

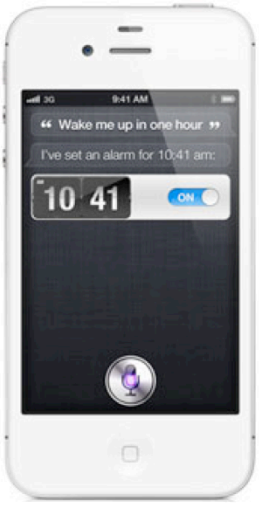
Error messages: CL, MF, and SD often have error messages (e.g. "you didn't enter a phone number"), but DM rarely needs error messages. There's no error message when you drag a scrollbar too far, for example; the scrollbar thumb simply stops, and the visual constraints of the scrollbar make it obvious why it stopped.

Efficiency: Experts can be very efficient with CL, since they don't need to wait for and visually scan system prompts, and many CL systems have command histories and scripting facilities that allow commands to be reused rather than constantly retyped. Efficient performance with MF interfaces demands good shortcuts (e.g. keyboard shortcuts, tabbing between form fields, typeahead). Efficient performance with DMs is possible when the DM is appropriate to the task; but using DM for a task it isn't well-suited for may feel like manual labor with a mouse.

User type: CL is generally better for expert users, who keep their knowledge active and who are willing to invest in training and learning in exchange for greater efficiency. MF, DM, and SD are generally better for novices and infrequent users.

Synchrony: Command languages are synchronous (first the user types a complete command, then the system does it). So are menu systems and forms; e.g., you fill out a web form, and then you submit it. Speech requires turn-taking between the system and user, so it's synchronous as well. DM, on the other hand, is asynchronous: the user can point the mouse anywhere and do anything at any time. DM interfaces are necessarily event-driven.

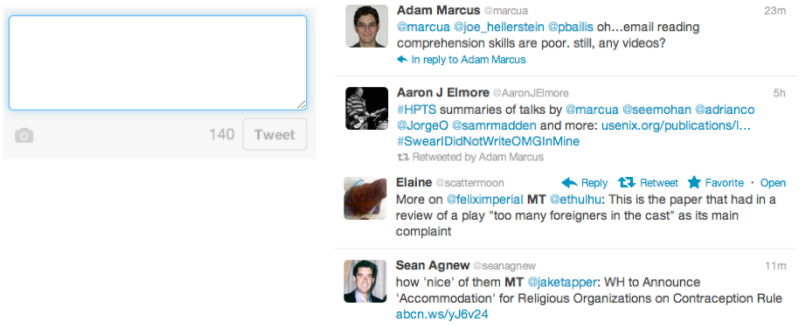
Using Multiple Interaction Styles



Real user interfaces often combine multiple interaction styles to make up for deficiencies in one style. For example, the Siri system built into iOS has both speech dialog (the user speaks something like “wake me up in one hour”, and the system replies with speech) and menu/form (the alarm time and on/off setting can be manipulated here).

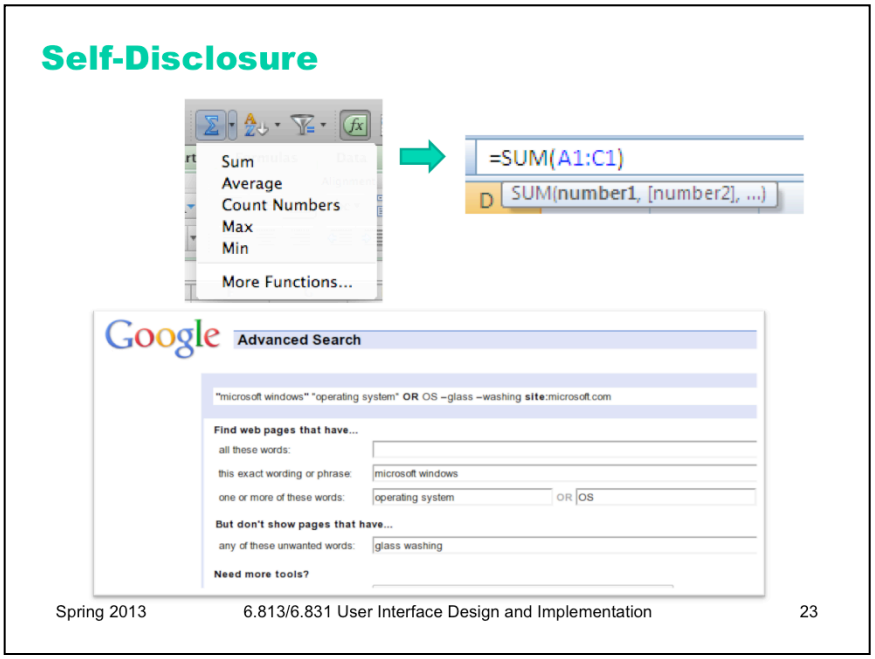
Example: Twitter's Tweet-Creation UI

What aspects of this UI use knowledge in the head?
What aspects use knowledge in the world?



Let's look at Twitter's interface – specifically, let's focus on the interface for creating a new tweet. What aspects of this interface are knowledge-in-the-world, and what aspects require knowledge in the head? In what way is Twitter a hybrid of a command language and a menu/form interface?

Twitter is actually an unusual kind of command interface in that examples of "commands" (formatted tweets generated by other users) are constantly flowing at the user. So the user can do a lot of learning by watching on Twitter. On the other hand, learning by doing is somewhat more embarrassing, because your followers can all see your mistakes (the incorrect tweets you send out while you're still figuring out how to use it).



Self-disclosure is a technique for making a command language more visible, helping the user learn the available commands and syntax. Self-disclosure is useful for interfaces that have both a traditional GUI (with menus and forms and possibly direct manipulation) as well as a command language (for scripting). When the user issues a command in the GUI part, the interface also displays the command in the command language that corresponds to what they did. A primitive form of self-disclosure is the address bar in a web browser – when you click on a hyperlink, the system displays to you the URL that you could have typed in order to visit the page. A more sophisticated kind of self-disclosure happens in Excel: when you choose the sum function from the toolbar, and drag out a range of cells to be summed, Excel shows you how you could have typed the formula instead. (Notice that Excel also uses a tooltip, to make the syntax of the formula more visible.)

On the bottom is another example of self-disclosure: Google's Advanced Search form, which allows the user to specify search options by selecting them from menus, the results of which are also displayed as a command-based query ("microsoft windows" "operating system" OR OS –glass –washing site:microsoft.com) which can be entered on the main search page. (example suggested by Geza Kovacs)

Picoquiz

Which of the following interaction styles uses knowledge-in-the-world heavily? (**choose all good answers**)

A. menus and forms
B. speech dialog
C. command languages
D. direct manipulation

To help you check your understanding and prepare for the nanoquiz in class, the lecture notes include a "picoquiz" question after every section. The picoquiz questions are similar to the nanoquiz questions you can expect to see in class, but you can check your answers yourself.

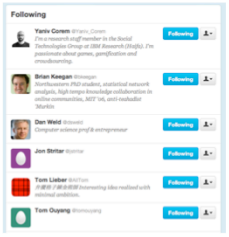
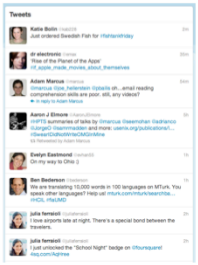
Note that the picoquiz has **all** the questions from this lecture, including questions from later in the lecture that you may not have read the material for yet. You can do each question as you get to it in your reading, and then submit the whole quiz when you've finished reading.

To answer the picoquiz questions in this lecture, go to:
http://courses.csail.mit.edu/6.831/2013/picoquiz?**lectureId=2**

CONCEPTUAL MODELS

Models

- **Model** of a system = how it works
  - its constituent parts and how they work together to do what the system does

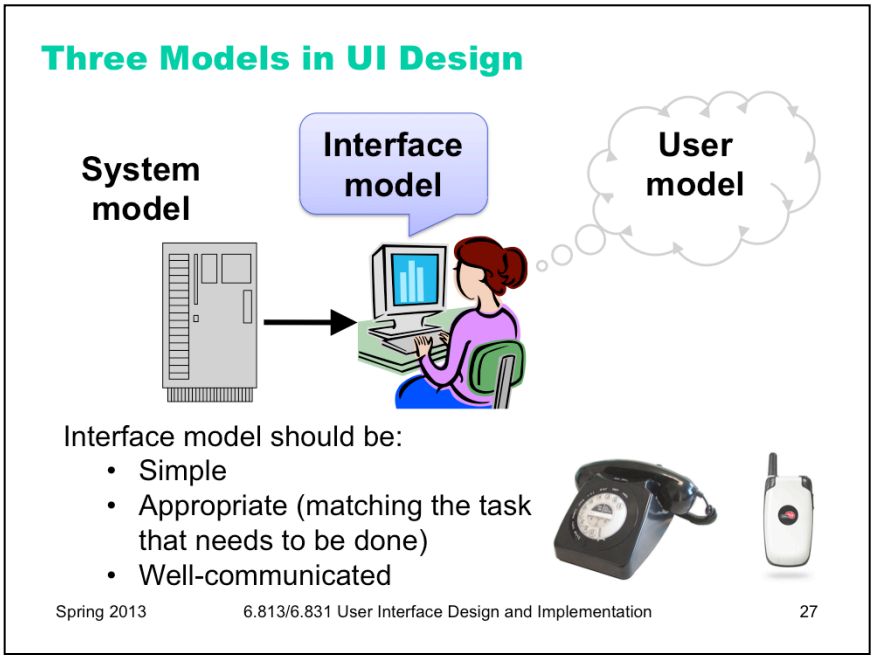
Follows   Tweets

Spring 2013 6.813/6.831 User Interface Design and Implementation 26

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.

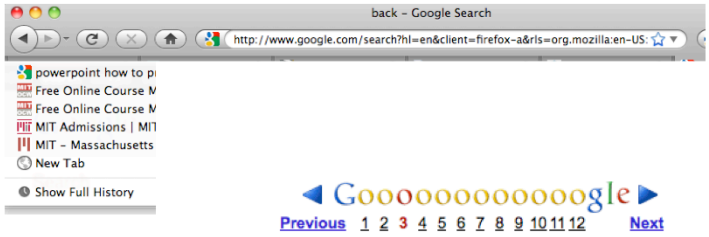
•The **user model** (or conceptual model) is how the user *thinks* the system works.

A cell phone presents the same simple interface model as a conventional wired phone, even though its system model is quite a bit more complex. A cell phone conversation may be handed off from one cell tower to another as the user moves around. This detail of the system model is hidden from the user.

As a software engineer, you should be quite familiar with this notion. A module interface offers a certain model of operation to clients of the module, but its implementation may be significantly different. In software engineering, this divergence between interface and implementation is valued as a way to manage complexity and plan for change. In user interface design, we value it primarily for other reasons: the interface model should be simpler and more closely reflect the user's model of the actual task.

Note that we're using *model* in a more general and abstract sense here than when we talk about the model-view-controller pattern (which you may have heard of previously, and which we'll discuss more in a future lecture). 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.

Example: Back vs. Previous

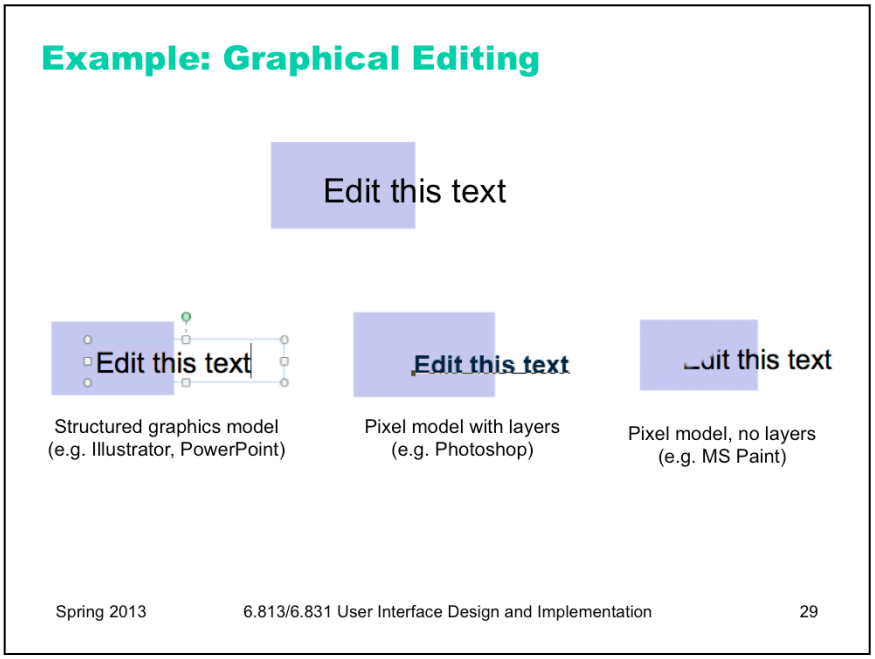


Here's an example drawn directly from graphical user interfaces: the Back button in a web browser. What is the model for the behavior of Back? Specifically: how does the *user* think it behaves (the mental model), and how does it *actually* behave (the system model)?

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!

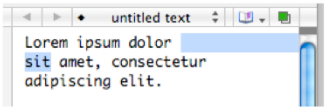
A nice article with other examples of tricky mental model/system model mismatch problems is "Mental and conceptual models, and the problem of contingency" by Charles Hannon, *interactions*, November 2008. http://portal.acm.org/citation.cfm?doid=1390085.1390099



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.

Example: Text Editing

Typewriter:
2D grid of characters



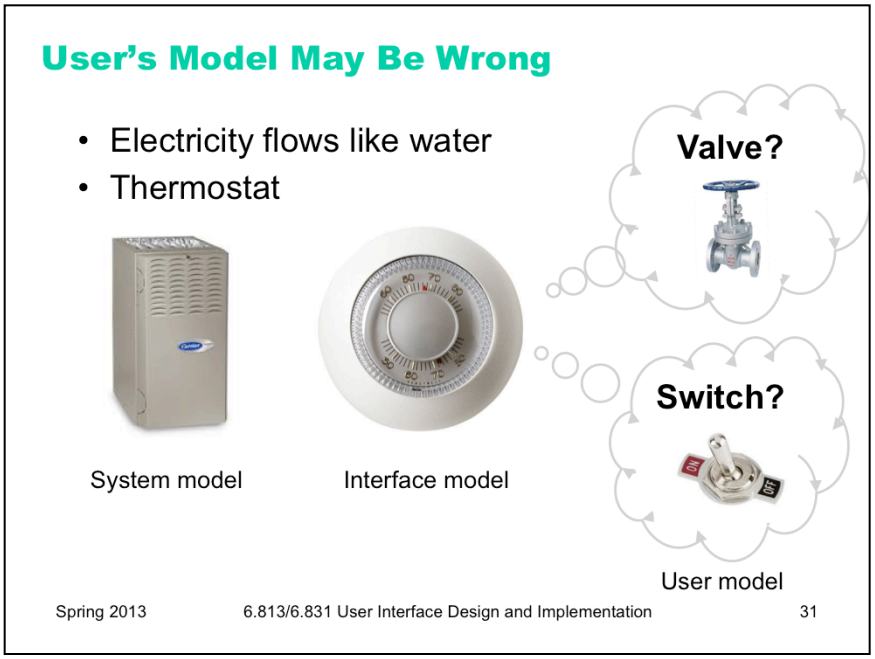
Text editor:
1D string with linebreak characters

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



The user's model may be totally wrong without affecting the user's ability to use the system. A popular misconception about electricity holds that plugging in a power cable is like plugging in a water hose, with electrons traveling from the power company through the cable into the appliance. The actual system model of household AC current is of course completely different: the current changes direction many times a second, and the actual electrons don't move far, and there's really a *circuit* in that cable, not just a one-way tube. But the user model is simple, and the interface model supports it: plug in this tube, and power flows to the appliance.

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.

Picoquiz

Which of the following statements are true? (**choose all good answers**)

A. Learning by doing is a primary way that users construct a mental model of a system.
B. The RealCD player's metaphor of a CD jewel case is an example of a system model.
C. The metaphor that electricity is like water is an example of a user model.
D. A good interface model should match the system model closely.

To help you check your understanding and prepare for the nanoquiz in class, the lecture notes include a "picoquiz" question after every section. The picoquiz questions are similar to the nanoquiz questions you can expect to see in class, but you can check your answers yourself.

Note that the picoquiz has **all** the questions from this lecture, including questions from later in the lecture that you may not have read the material for yet. You can do each question as you get to it in your reading, and then submit the whole quiz when you've finished reading.

To answer the picoquiz questions in this lecture, go to:
http://courses.csail.mit.edu/6.831/2013/picoquiz?**lectureId=2**