Lecture 6:
Models & Metaphors

UI Hall of Fame or Shame?

Source: Interface Hall of Shame

Fall 2004   6.831 UI Design and Implementation   2

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

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

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

There's an obvious problem with the choice of metaphor, of course: a CD case doesn't actually play CDs. The designers had to find a place for the player controls – which, remember, serve the primary task of the interface – so they arrayed them vertically along the case hinge. The metaphor is dictating control layout, against all other considerations.
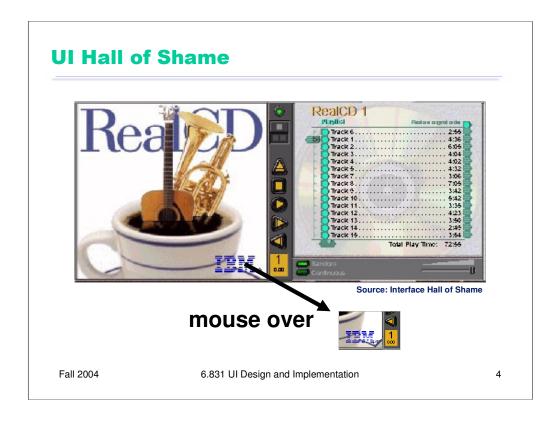
Slavish adherence to the metaphor also drove the designers to disregard all consistency with other desktop applications. Where is this window's close box? How do I shut it down? You might be able to guess, but is it "intuitive?" Learnability comes from more than just metaphor.

2

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?

3

UI Hall of Shame

Source: Interface Hall of Shame

mouse over

We're not done yet. Where is the online help for this interfaceand its method of activation is again dictated by the metaphor.

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

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

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

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

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

Today's Topics

- Conceptual models
- Interaction styles
- Direct manipulation
- Errors
- Metaphors

Today's lecture concerns the **conceptual models** of user interfaces. A metaphor, like the CD case, is an example of a conceptual model. In a sense, your job as a user interface designer boils down to (1) choosing the right conceptual model and (2) teaching it successfully to the user.

We'll discuss some techniques for successful communication, among them affordances, mapping, visibility, and feedback. We already encountered **affordances** in the first lecture, when we discussed the Hall of Shame award-certificate printing program, which used a scrollbar in a way contrary to its affordance.

We'll also discuss what causes users to make **errors**, even after all the UI tries to tell them, and how we can design our systems to prevent or mitigate these errors. An important kind of error is caused by **modes**. We'll see what modes are and how to avoid them.

Finally, we'll come back to **metaphors** again, and try to understand when they might help, and when not.

Models

- **Model** of a system = how it works
  - its constituent parts and how they work together to do what the system does
- Implementation models
  - Pixel editing vs. structured graphics
  - Text file as single string vs. list of lines
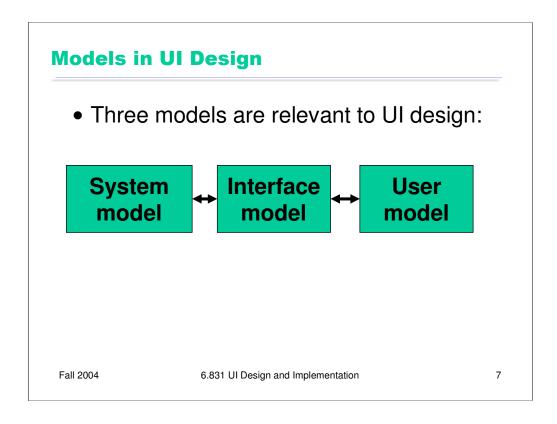- Interface models
  - RealCD's online help as liner notes

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.

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 Visio 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 Photoshop, 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 the *vi* text editor, line endings can't be deleted in the same way as other characters. *Vi* has a special join command for deleting line endings.

A model may concern only a small part of a system. For example, RealCD has a model for its online help that imitates a CD's liner notes.

Models in UI Design

- Three models are relevant to UI design:

System model	↔ **Interface model** ↔	**User model**

The preceding discussion hinted that 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.

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

Interface Model Hides System Model

- Interface model should be:
  - Simple
  - Appropriate: reflect user's model of the task (learned from task analysis)
  - Well-communicated

The interface model might be quite different from the system model. A text editor whose system model is a list of lines doesn't have to present it that way through its interface. The interface could allow deleting line endings as if they were characters, even though the actual effect on the system model is quite different.

Similarly, 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, which we can learn from task analysis.

User Model May Be Wrong

- ## Sometimes harmless
  - Electricity as water
- ## Sometimes misleading
  - Thermostat as a valve

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 up 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 much. 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: the heater in a car, for example, or a burner on a stove. 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.

Interaction Styles

- Command language
- Menus & forms
- Direct manipulation

Let's get a little more concrete now, and look at three major kinds of user interface styles. We'll tackle them in roughly chronological order as they were developed.

Command Language

- User types in commands in an artificial language
- Examples
  - Unix shell ("ls −l *.java")
  - Search engine query language ("AND, OR, site:www.mit.edu")
  - URLs ("http://www.mit.edu/admissions/")
- Command syntax is important

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.

When designing a command language, the key problem is designing the command syntax. Task analysis drives the choice of commands, the names you give them, the parameters they have, and the syntax for fitting them together.

Menus and Forms

- User is prompted to choose from menus and fill in forms
- Examples
  - virtually all web sites
  - dialog boxes
- Navigation structure is important
  - Menu trees (Yahoo!)
  - Wizard: linear sequence of forms

A menu/form interface presents a series of menus or forms to the user. Virtually all web sites behave this way. Dialog boxes are a form-style interface frequently found embedded inside another interaction style.

The navigation structure is the important design problem for menu/form interfaces. Task analysis tells you what choices need to be available, where they should be placed in a menu tree, and what data types or possible responses need to be available in a form.

Direct Manipulation

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

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

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.

Comparison of Interaction Styles

- Knowledge in the head vs. world
- Error messages
- Efficiency
- User experience
- Synchrony
- Programming difficulty
- Accessibility

Error messages: DM rarely needs them. No error message when you drag a scrollbar "too far", for example.

Knowledge in the head vs. the world. Command languages require the user to put a lot of knowledge into their heads, by training, practice, etc. (Or else compensate by having manuals, reference cards, or online help close at hand while using the system.) Menus and forms put much more information into the world. Well-designed DM also has information in the world, from the affordances and constraints of the visual metaphor.

Efficiency: experts can be very efficient with command languages (no need to scan system prompts; able to reuse commands in scripts and history). Efficient performance with menus and form 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 (and depends on shortcuts and Fitts's Law); but using a DM for a task it isn't designed for may turn into manual labor.

User experience: command languages best for expert users – frequent, well-trained. Menus/forms and DMs 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. web model. DM is asynchronous: user can point mouse anywhere and do anything at any time.

Programming difficulty: Command languages are relatively easy: parsing text, rigid requirements. Menus and forms have substantial toolkit support; e.g., the web browser and HTML, Java Swing components. DM is hardest to program: have to draw, handle low-level input (keyboard, mouse), display feedback. Very few off-the-shelf components available to help. DM can't be done easily on the Web without Java or Flash.

Accessibility: command and menu/form interfaces are more textual, so easier for vision-impaired users to read with screen readers. DM interfaces are much harder for these users.

Direct Manipulation Cues

- Affordances
- Constraints
- Natural mapping
- Visibility
- Feedback

So what is the language by which a direct manipulation interface communicates its model to the user? Or, looking at it from the user's perspective, what cues do users rely on in order to learn the model: the **parts** that make up the interface, and **how those parts work together**?

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

Affordances

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

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

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

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

Natural Mapping

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

Logical constraints lead to another important principle of interface communication: **natural mapping** of functions to controls.

Consider the spatial arrangement of 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, it is much easier to learn which switch controls which light.

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

Other good examples of mapping include:

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

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

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

Visibility

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

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

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

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

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

Feedback

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

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

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

Modeling Human Error

- Description error
- Capture error
- Mode error

Let's say a bit about some of the kinds of errors that humans make.

Description Error

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

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

To limit description errors in computer interfaces, avoid actions with very similar descriptions, like long rows of identical switches.

Capture Error

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

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

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

Mode Error

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

A third kind of error is a mode error. **Modes** are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands.

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

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. When modes are necessary, it's essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn't actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user's locus of attention. That's why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn't really work. (Raskin, The Humane Interface, 2000 has a good discussion of locus of attention as it relates to mode visibility.)

Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do something active to stay in the alternate mode, essentially eliminating the chance that they'll forget what mode they're in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another spring-loaded mode; you're only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette, that drawing mode is active only for one mouse gesture. Once you've drawn one rectangle, the mode automatically reverts to ordinary pointer selection.

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

Metaphors

- Another way to address the model problem
- Examples
  - Desktop
  - Trashcan

Let's close by looking again at metaphors. We started out by talking about RealCD's, an example of an interface that uses a metaphor for its interface model.

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 and folders on a desk-like surface – is widely used and copied. The trashcan, a place for discarding things but 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?).

Dangers of Metaphors

- Hard to find
- Deceptive
- Constraining
- Breaking the metaphor

- Use of a metaphor doesn't excuse bad communication of the model:
  – RealCD's bad affordances, visibility

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), so 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.

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

Most of all, use of a metaphor doesn't excuse an interface that does a bad job communicating its model 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.