

6.813/6.831 User Interface Design

General Information

6.813/6.831 introduces the principles of user interface development, focusing on the following areas:

- **Design.**
We will look at how to design good user interfaces, covering important design principles (consistency, visibility, simplicity, efficiency, and graphic design) and the human capabilities that motivate them (including perception, motor skills, color vision, attention, and human error).
- **Implementation.**
We will see techniques for building user interfaces, including low-fidelity prototyping, input, output, model-view-controller, and layout.
- **Evaluation.**
We will learn techniques for evaluating and measuring interface usability, including heuristic evaluation, predictive evaluation, and user testing.
- **Research (6.831 only).**
We will learn how to conduct empirical research involving novel user interfaces.

Course material will include lecture notes and assigned readings from the Web.

Web Site

The course web site is on Stellar. Jump to it quickly by Googling for 6.831.

Staff

Contact information for the teaching staff is found on the web site.

Credit

3-0-9 = 12 units (U credit for 6.813, graduate H credit for 6.831)

6 Engineering Design Points

For EECS undergraduates, 6.813 can satisfy either the Advanced Undergraduate Subject requirement or the department software lab requirement.

For EECS PhD students, 6.831 satisfies the TQE requirement in either Systems or AI, but only as a second subject, not as the sole subject in the area.

Prerequisites

6.005 or equivalent undergraduate software engineering experience.

Grading

The largest contribution to your grade will be the course project (42%), in which you will work in small groups to design, implement, and evaluate a user interface, through an iterative design process with a series of graded milestones (GR1-GR6). Students from 6.813 and 6.831 may work in the same group.

Five problem sets (HW, PS/RS) will be assigned, which you must complete individually, not in a group. HW1-2 ("homeworks") are assigned to both courses; PS1-3 ("programming") are assigned only to the undergraduate course 6.813; and RS1-3 ("research") are assigned only to the graduate course 6.831. These five assignments will constitute 30% of your grade.

Every lecture will begin with a "nanoquiz," which covers the content of the previous lecture or two. There will be approximately 30 nanoquizzes, which altogether count for 24% of your grade. If you miss class, no makeup quiz is offered. However, we will automatically drop your lowest 6 quiz grades, so that you have flexibility to miss class when necessary. We also offer a way to make up low nanoquiz grades; see Stellar for a link to the nanoquiz makeup submission form.

There will be no other in-class quizzes, no midterm, and no final exam.

Participation in lecture, online forums, in-class activities, and project group meetings with course staff will also be a factor in your grade (4%).

| | |
|--------------------------|-----|
| Course project (GR1-6) | 42% |
| Problem sets (HW, PS/RS) | 30% |
| Nanoquizzes | 24% |
| Class participation | 4% |

Collaboration

You may discuss assignments with other people, but you are expected to be intellectually honest and give credit where credit is due. In particular, for all individual assignments (HW, PS, RS):

- you should write your solutions entirely on your own;

- you should not share written materials or code with anyone else;
- you should not view any written materials or code created by anyone else for the assignment; and
- you should list all your collaborators (everyone you discussed the assignment with) on your handin.

Failure to adhere to these policies may result in serious penalties, up to and including automatic failure in the course and reference to the Committee on Discipline.

Lateness and Extensions

To give you some flexibility for periods of heavy workload, minor illness, absence from campus, and other unusual circumstances, you may request limited extensions on problem set deadlines, called slack days. Each slack day is a 24-hour extension on the deadline. You have a budget of 5 slack days for the entire semester, which you may apply to any combination of individual assignments (HW1-2, PS1-3, or RS1-3). You can use at most 3 slack days for a given assignment. Assignments more than three days late will not be accepted.

You must request your extension before the problem set is due, by using the **Slack Day Request System**. The system keeps track of your slack days and informs you how many you have left.

Slack days apply only to individual problem sets, not to group work (GR1-GR6). For group assignments, we will grade whatever you have on your wiki at handin time.

If you have used up your slack days, or exceeded the 3-day limit for a single assignment, you will need a lecturer's permission and support from an S&3 dean for more extension.

Differences between 6.813 and 6.831

Students must choose between the undergraduate course 6.813 and the graduate course 6.831. Each version satisfies different requirements in the various EECS degree programs; see your degree program for more details. This section summarizes the main differences between the two courses.

In general, the graduate version is a strict superset of the undergraduate version.

Course content. Students in the graduate course are responsible for all the material in the undergraduate course (lectures on design, implementation, and evaluation), *plus* additional material (lectures on research methods). Some nanoquizzes will include extra questions only for the graduate course.

Assignments. The undergraduate problem sets PS1-PS3 cover implementation techniques. The graduate course's RS1-RS3 cover research methods. Both courses share the same HW1 and HW2.

Group project. Both courses have the same group project, and students from either course may freely work together in the same group.

Since the two courses have substantial overlap, MIT will allow you to get credit for *only one* of them during your MIT career. Keep this in mind when deciding which course is right for you. You can change your mind and switch to the other course any time before Add Date (by dropping one number and adding the other), but not thereafter. In any case, you are responsible for all the requirements of the course you finally register for.

Textbooks

There is no required textbook.

Recommended books:

- Norman, *The Design of Everyday Things*, 1990.
This little book is a classic work on usability, not just of computer interfaces but also of physical objects like doors, showers, and stoves. Full of great anecdotes, plus theory about how users form models in their heads and how users make errors. Belongs on every engineer's shelf.
- Nielsen, *Usability Engineering*, Academic Press, 1993.
Somewhat dated but still useful handbook for discount usability engineering, covering many of the evaluation techniques we'll be learning in this class.
- Mullet & Sano, *Designing Visual Interfaces*, Prentice Hall, 1995.
A terrific guide to graphic design, chock full of examples, essential principles, and practical guidelines. Many programmers have a fear of graphic design. This book won't teach you everything—it still pays to hire a designer!—but it helps get over that fear and do a competent job of it yourself.

Good references:

- Baecker, et. al., *Readings in Human-Computer Interaction: Toward the Year 2000*, Morgan Kaufmann, 1995.
- Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 4th ed., Addison-Wesley, 2004.
- Dix et al, *Human-Computer Interaction*, 2nd ed., Prentice-Hall, 1998.
- Olsen, *Developing User Interfaces*, Morgan Kaufmann, 1998.

Other books we like:

- Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1983.

- Raskin, The Humane Interface , ACM Press, 2000.
- Johnson, GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers , Morgan Kaufman, 2000.
- Card, Moran, & Newell, The Psychology of Human-Computer Interaction , Lawrence Erlbaum, 1983.

Books about statistics and experiment design:

- Gonick, Cartoon Guide to Statistics , Harper, 1994.
- Box, Hunter, & Hunter. Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building , Wiley, 1978.
- Miller, Beyond Anova: Basics of Applied Statistics , Wiley, 1986.

Michael E Plasmeier

From: Rob Miller <rcm@MIT.EDU>
Sent: Monday, January 30, 2012 11:20 PM
To: Robert C Miller
Subject: 6.813/6.831 announcement: 6.813/6.831: mandatory first-day attendance and 6.005 prerequisite

Note: This mail was sent to all students in the stellar class User Interface Design

6.813/6.831: mandatory first-day attendance and 6.005 prerequisite

You're getting this message because you preregistered for 6.813/6.831 User Interface Design and Implementation for the spring semester. We're looking forward to a fun class this semester, but there are a few things you need to know.

1. Attendance is required at the first lecture on Wed Feb 8 (1-2 pm in 34-101). If you're not there, you won't be able to take the course.
2. Laptops will be required for lectures -- including the first. If you don't have a functioning laptop, IS&T has a loaner program (<http://ist.mit.edu/services/hardware/lcp>), but you should act now to get one.
3. The course has a prerequisite: 6.005 or equivalent software engineering experience. In particular, we expect you to have all of the following:
 - (a) experience developing software in a group of 3-5 people;
 - (b) proficiency in a software-engineering language, such as Java, C++, C#, or Objective C;
 - (c) experience implementing at least one graphical user interface in one of these languages.

Note that programming experience limited to HTML, Javascript, or Python does NOT satisfy the prerequisite for this course, and you will not be able to enroll.

When you fill out the signup form at the first lecture, you will have to explain how you satisfy the prerequisite.

That's it for now. See you next week!

This announcement was made in Stellar on 2012 January 30 by Rob Miller

The announcement is also posted on the class website:

<https://stellar.mit.edu/S/course/6/sp12/6.813/>

(much more cranked than I thought!)

Gave at combo labs

Prof Rob Miller

(Filled out sign up sheet)

Usability

- property of a UI
- UIs could be lots of things

Certificate UI Hall of Shame

Long help text

Scroll bar looks like panning over image

But is going between certs - bad!

- non standard
- discrete selection
- hard to see
- Learnability
- Efficiency
- Usability

② Time

Affordances - way display appears

Text box that rejects key strokes

Different representations of time

Clicking advances it | stop

GIMP

Learnability problem

How to know to learn

But it actually makes it slower

Since need to move down narrow tunnel

If you hit a key when menu is open → it reassigns menu item

↳ Mode issue - mode changes

accidents waiting to happen - safety

3rd principle of usability

~~Emacs~~ Emacs

Incremental search

↳ everyone does this now

But you have to know Ctrl+S is the keyboard shortcut

③

Dialog box covers text to search

But is modellless - so can edit text when box is open

- but you would not know from the view

Clippy

Was good intention - teach you as you use the program

Since you don't use manual

But he interrupts

Users

Users are not you or not like you

Usability issues is program's issue - not w/ user

Usability is not always the problem (only time he will say that)

Users are not always right w/ what they want

- Telephone weight

- # Google results

④

Usability = how well users can use system's functionality

- └ Learnability - 'is it easy to learn'
- Efficiency - once it's learned, is it fast to use
- Safety - are errors few + recoverable

Other

Ergonomics - comfort, fatigue

Aesthetics - satisfaction, happiness, pleasure

Can quantify by testing

└ not just a fluffy subject

Importance of each factor depends on application

Usability is only 1 aspect of a system

- functionality
- performance
- cost
- security
- dependability
- etc

We'll take an extreme position in this class

5

Think about usability of a combo lock

- good: can't change combo - so can't accidentally change it to something you will forget...

Learnability

- needs instructions
- are they well written?
- pretty consistent across brands
- need to know clock conventions
- simple feature set
- small notch shows state
- need to learn protocol + H's
 - could make mistake in either one
 - no feedback which one

Efficiency

- slow to open (good for security)
- need to restart if make mistake
- muscel memory?

Safety

- no feedback
- code stuck too tight on back
- might need extra step to relax
- need to turn after locking or show H

⑥

What you'll learn

- design principles
- design techniques
- implementation techniques
- grad class → research methods

Nanoquizzes online in class each time

L1: Usability

Spring 2012

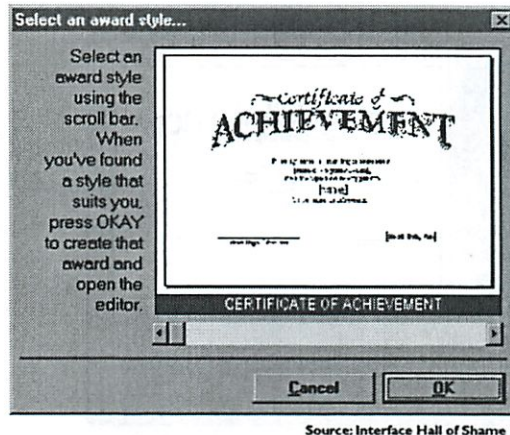
6.813/6.831 User Interface Design and Implementation

1

Today's Topics

- UI Hall of Fame and Shame
- Why UI design is hard
- Usability defined
- Course overview

User Interface Hall of Shame



Spring 2012

6.813/6.831 User Interface Design and Implementation

3

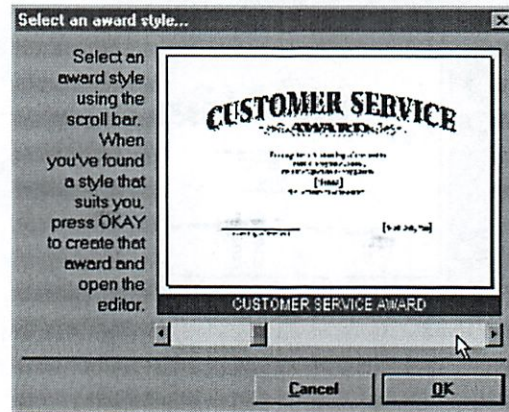
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template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

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

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

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

User Interface Hall of Shame



Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface Design and Implementation

4

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.

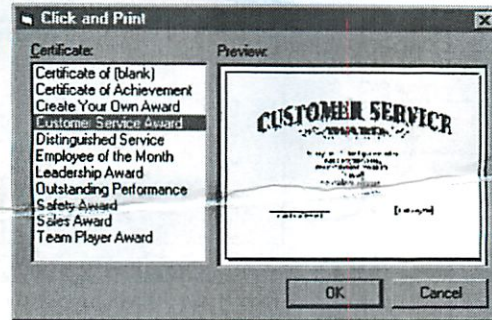
But the dialog doesn’t get any better for frequent users, either. If a frequent user wants a template they’ve used before, how can they find it? Surely they’ll remember that it’s 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes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

The Example, Redesigned



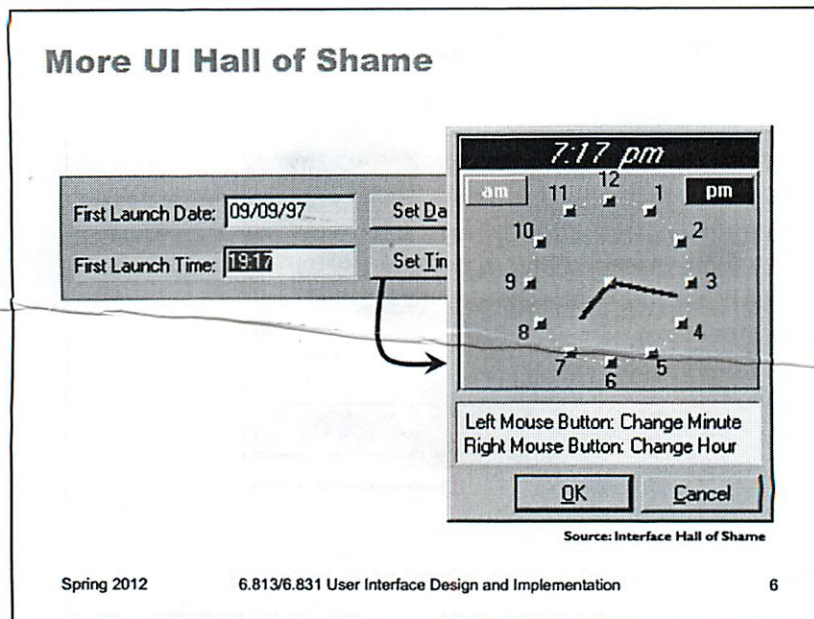
Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface Design and Implementation

5

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.



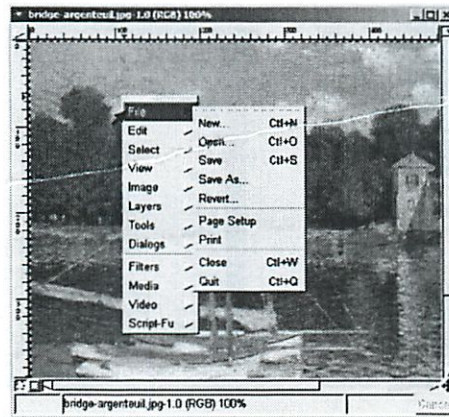
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to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

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

7

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.

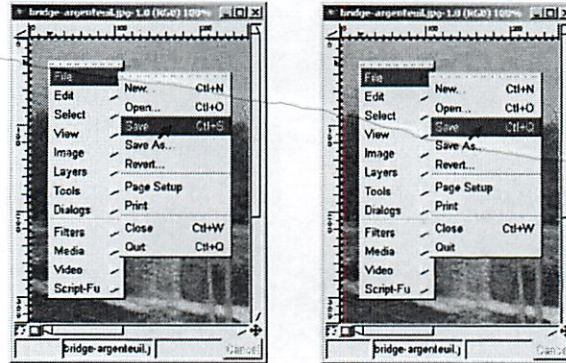
This is certainly inconsistent with other applications, and new users are likely to stumble trying to find, for example, the File menu, which never appears on a context menu in other applications. (I certainly stumbled as a new user of Gimp.) But Gimp's designers were probably thinking about expert users 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. 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! Hierarchical submenus are actually slower to use than a menu bar.

Part of the problem here is the way GTK (the UI toolkit used by Gimp) implements submenus. Changing the submenu immediately is probably a bad idea. 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 you still have to make that right-angle turn. 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. So you can drive diagonally toward Quit without losing the File menu, or you can drive straight down to get to the Edit menu instead.

Gimp's designers made a choice without fully considering how it interacted with human capabilities. We'll see that there are some techniques and principles that we can use to predict how decisions like this will affect a user interface – and we'll also see how we can measure and evaluate the actual effects.

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

8

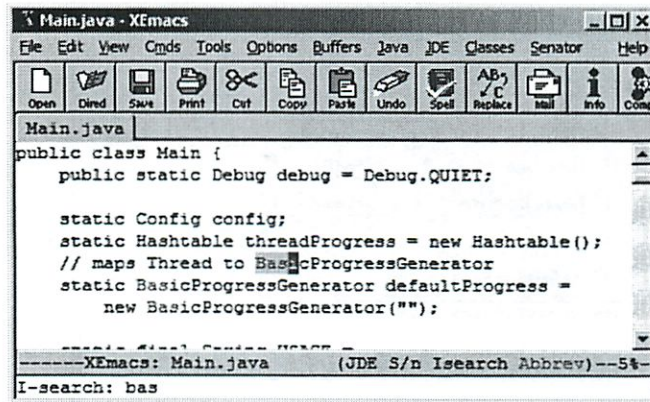
There's another interesting design feature in Gimp's menus --- well-intentioned and clever, but problematic in practice. Suppose my mouse is halfway down the File menu when I notice that the Quit command actually has a keyboard shortcut: Ctrl-Q. Great! So I press it.

But it doesn't invoke the Quit command. Instead, it changes the shortcut of whatever command my mouse is hovering over --- in this case, Save --- to Ctrl-Q. This is a **mode**: a state of the system in which a user action has a different meaning than it does in other states. Modes may be inevitable in user interfaces, but **mode errors** --- using the action in the wrong mode, so it does something you don't intend --- do not have to be inevitable.

Worse, it's not an easy error to undo. (Pressing Ctrl-Z, the conventional undo shortcut, only makes it worse!) I have to reassign the old shortcut to the Save command --- if I can remember the original shortcut. Then I have to find the command whose original shortcut was Ctrl-Q, and restore that one as well. This error wasn't easily **recoverable**.

Gimp's designers had a terrific idea here --- making it easy to assign keyboard shortcuts by just pointing at the menu item you want to change and pressing the shortcut. That's simple and elegant, in fact far simpler than most customization interfaces. But they've given us too much rope, and it's easy to hang ourselves. This interface isn't **safe** to use.

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

9

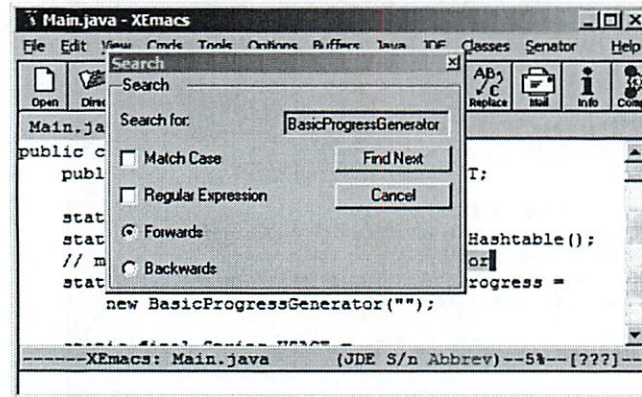
In Emacs, Ctrl-S starts an *incremental search*. This is a well-designed feature.

- it's highly **responsive**: updates as fast as the user can type
- it's easily and obviously **reversible**: press Backspace if you made a mistake
- it provides immediate **feedback** about what it's doing
- successful searches may even achieve early success: only 3 letters was enough to find BasicProgressGenerator, and I could instantly tell that it was enough
- user gets early feedback about typos and failed searches

What's the downside? All its controls are **invisible**. How do you start the incremental search? How do you search again? How do you go backwards? How do you do a case-sensitive search?

Once learned, however, these commands are simple. Ctrl-S starts the search. Pressing Ctrl-S again looks for a later match. (But now there is the possibility of mode error!) Pressing Ctrl-R looks backwards for a previous match. (What does Backspace do?) Using any capitalized letters in your query forces a case-sensitive search. (But how do I search for an all-lowercase string case-sensitively?)

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

10

XEmacs has menus (the original Emacs didn't). Alas, XEmacs isn't interested in helping users learn incremental search. Instead, it pops up this conventional Find dialog, which scores great on visibility, but lacks the responsiveness, easy reversibility, and fast performance of incremental search. Even worse, it covers up the matches you're trying to find unless you manhandle it out of the way!

UI Hall of Fame or Shame?

It looks like you're writing a letter.
Would you like help?

- Get help with writing the letter
- Just type the letter without help

☐ Don't show me this tip again



Clippy is sad.

Are you sure you want to kill off the Clippy?

Yes

No



Spring 2012

6.813/6.831 User Interface Design and Implementation

11

Finally, we have the much-reviled Microsoft Office Paperclip.

Clippy was a well-intentioned effort to solve a real usability problem. Users don't read the manual, don't use the online help, and don't know how to find the answers to their problems. Clippy tries to suggest answers to the problem it thinks you're having.

Unfortunately it's often wrong, often intrusive, and often annoying. The subjective quality of your interface matters too.

You Are Not the User

- Most software engineering is about communicating with other programmers
...who are a lot like you
- UI is about communicating with users
 - Users are NOT LIKE YOU
- The user is ALWAYS right
 - Usability problems are the design's fault

```
/**
 * simple HelloWorld() method.
 *
 * @version 1.0
 * @author John Doe <doe.j@example.com>
 */
public class HelloWorld {
    public static void main(String[] args) {
        // use the JFrame type until support for 1
        // new component is finished
        JFrame frame = new JFrame("Hello World")
        Container pane = frame.getContentPane();
        pane.add( hello );
        frame.pack();
    }
}
```



Spring 2012

6.813/6.831 User Interface Design and Implementation

12

Unfortunately, user interfaces are not easy to design. You (the developer) are not a typical user. You know far more about your application than any user will. You can try to imagine being your mother, or your grandma, but it doesn't help much. It's very hard to *forget* things you know.

This is how usability is different from everything else you learn about software engineering. Specifications, assertions, and object models are all about communicating with other *programmers*, who are probably a lot like us. Usability is about communicating with other *users*, who are probably not like us.

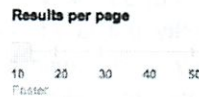
The user is always right. Don't blame the user for what goes wrong. If users consistently make mistakes with some part of your interface, take it as a sign that your *interface* is wrong, not that the users are dumb. This lesson can be very hard for a software designer to swallow!

Don't Expect Users to be Designers

- Telephone handset weight
 - Users said: it's fine
 - But they really wanted: lighter



- # of Google search results
 - Users said: 30 results
 - But they really wanted: 10



- Command abbreviations
 - Users make 2x errors with their own custom abbreviations

```
ls  cp
rm  cat
mv
```

Spring 2012

6.813/6.831 User Interface Design and Implementation

13

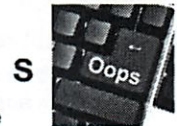
Unfortunately, the user is not always right. Users aren't oracles. They don't always know what they want, or what would help them. In a study conducted in the 1950s, people were asked whether they would prefer lighter telephone handsets, and on average, they said they were happy with the handsets they had (which at the time were made rather heavy for durability). Yet an actual test of telephone handsets, identical except for weight, revealed that people preferred the handsets that were about half the weight that was normal at the time. (Klemmer, *Ergonomics*, Ablex, 1989, pp 197-201).

Another example: Google has discovered that when they survey users about how many search results they want per page (10, 20, 30), users overwhelmingly say "30 results". But when Google actually *deploys* 30-result search pages (as part of an "A/B test", which we'll talk about in a later lecture), usage drops by 20% relative to the conventional 10-result page. Why? Probably because the 30-result page takes a half second longer to load. (<http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx>)

Users aren't designers, either, and shouldn't be forced to fill that role. It's easy to say, "Yeah, the interface is bad, but users can customize it however they want it." There are two problems with this statement: (1) most users don't, and (2) user customizations may be even worse! One study of command abbreviations found that users made twice as many errors with their *own* command abbreviations than with a carefully-designed set (Grudin & Barnard, "When does an abbreviation become a word?", CHI '85). So customization isn't the silver bullet.

Usability Defined

- **Usability** = how well users can use the system's functionality
- Dimensions of usability
 - **Learnability**: is it easy to learn?
 - **Efficiency**: once learned, is it fast to use?
 - **Safety**: are errors few and recoverable?
- Other dimensions are relevant too
 - Ergonomics: comfort, fatigue
 - Aesthetics: satisfaction, happiness, pleasure
 - But we'll mostly focus on LES



Spring 2012

6.813/6.831 User Interface Design and Implementation

14

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 safety. These aren't the only aspects of a user interface that you might care about (for example, subjective feelings are important too, as is fatigue), but these are the primary ones we'll care about in this class.

Notice that we can **quantify** all these measures of usability. Just as we can say algorithm X is faster than algorithm Y on some workload, we can say that interface X is more learnable, or more efficient, or more safe than interface Y for some set of tasks and some class of users, by designing an experiment that measures the two interfaces.

Usability Dimensions Vary In Importance

- Depends on the user
 - Novice users need learnability
 - Experts need efficiency
 - But no user is uniformly novice or expert
- Depends on the task
 - Missile launchers need safety
 - Subway turnstiles need efficiency



Spring 2012

6.813/6.831 User Interface Design and Implementation

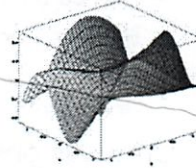
15

The usability dimensions are not uniformly important for all classes of users, or for all applications. That's one reason why it's important to understand your users, so that you know what you should optimize for. A web site used only once by millions of people – e.g., the national telephone do-not-call registry – has such a strong need for ease of learning, in fact zero learning, that it far outweighs other concerns. A stock trading program used on a daily basis by expert traders, for whom lost seconds translate to lost dollars, must put efficiency above all else.

But users can't be simply classified as novices or experts, either. For some applications (like stock trading), your users may be *domain* experts, deeply knowledgeable about the stock market, and yet still be novices at your particular application. Even users with long experience using an application may be novices or infrequent users when it comes to some of its features.

Usability Is Only One Attribute of a System

- Software designers have a lot to worry about:
 - Functionality
 - Performance
 - Cost
 - Security
 - Dependability
 - Usability
 - Size
 - Reliability
 - Standards
 - Marketability
- Many design decisions involve tradeoffs among different attributes
- We'll take an extreme position in this class



Spring 2012

6.813/6.831 User Interface Design and Implementation

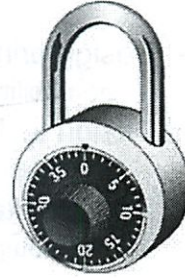
16

Usability doesn't exist in isolation, of course, and it may not even be the most important property of some systems. Astronauts may care more about reliability of their navigation computer than its usability; military systems would rather be secure than easy to log into. Ideally these should be false dichotomies: we'd rather have systems that are reliable, secure, *and* usable. But in the real world, development resources are finite, and tradeoffs must be made.

In this class, we'll take an extreme position: usability will be our primary goal.

Thinking about Usability

- Play with this device's UI
- Think about its usability
- Talk about it with your neighbors
- List its **good** and **bad** aspects on each usability dimension:
 - Learnability: is it easy to learn?
 - Efficiency: once learned, is it fast to use?
 - Safety: are errors few and recoverable?



Spring 2012

6.813/6.831 User Interface Design and Implementation

17

Here's a familiar UI design. You've probably used it before, but probably without thinking much about its usability. We're handing some out for you to play with and think about.

What You'll Learn in 6.813/6.831

- Design principles
 - learnability, visibility, errors, efficiency, ...
- Design techniques
 - task analysis, prototyping, user testing, ...
- Implementation techniques
 - MVC, output, input, layout, ...
- Research methods (6.831G only)
 - experiment design & analysis

Spring 2012

6.813/6.831 User Interface Design and Implementation

18

This course will be structured as four threads of lectures: design principles; design techniques; implementation techniques; and research methods. You have to gain experience in the first three in order to do your group project, which takes the entire semester, and the fourth is necessary for the 6.831 version of the course, which introduces graduate students to research in HCI.

Each lecture will be accompanied by lecture notes available on the course web site.

What You'll Get From 6.813/6.831

- A sense for usability
 - some knowledge of human capabilities
 - design principles and patterns for better usability
- A process for building usable interfaces
 - cheap prototypes
 - early and regular feedback from users
 - iterative design
- Experience with GUI implementation
 - HTML/Javascript
- (6.831G) Preparation for HCI research
 - controlled experiments
 - current HCI research topics

Spring 2012

6.813/6.831 User Interface Design and Implementation

19

Here are the key objectives of the course. This is what I hope you'll take away from the class.

Course Structure

- See course website for:
 - Lecture notes
 - Grading policy
 - Calendar
 - Group project
 - Problem sets & homeworks
 - Automatic extension policy
- Use Piazza to ask questions
- Collaboration policy

Spring 2012

6.813/6.831 User Interface Design and Implementation

20

Administrivia can be found on the course web site.

Nanoquizzes

- Every lecture will start with a 3-minute quiz
 - covers recent course material
 - taken online ←
 - we'll discuss the answers right after
 - your 6 lowest quiz grades will be discarded
 - makeup options are available on Stellar
- Simulated question
 1. Clippy is: (choose all answers that apply)
 - A. Annoying to many users
 - B. A paperclip
 - C. A violation of the usefulness dimension of usability
 - D. No longer in existence

Bring
your
laptop!

Next Time: Learnability



Source: Interface Hall of Shame

Find a team

Or use the Find a ~~team~~ team page

Hall of Shame

tried to make it like a ^{metaphor} jewel case

but vertical - rather than ~~old~~ traditional horiz

- screwed up metaphor

large unclickable areas

Open jewel case

- now buttons in the middle

No recognisable scroll bar

Get liver notes out when mouse over corner

↳ feedback - good

But program help

Nano Quiz

(difficult to understand what they want...)

①

Quantitative + objective

Security is not a usability dimension
Cost
dependability

learnability is a usability dimension
efficiency
safety

efficiency not imp in a small homeless shelter
safety is - want less mistakes
most important

expert users - learnability not important
lots of grades
Error free

Config interface - learnability
not efficiency - just 1
not safety - does not care about paper + toner
Miller: turn it off

③

(Rob Miller is very particular on multiple choice - annoying)

How we learn UI

- not manual

- not training

- not reading online help

) in general

Users have a task/goal

Users explore program trying to find right feature

When use help - have a specific problem

- so search box is good

Lots of people use google

Can use Google Instant suggestions to see top problems

Can see where learnability is breaking down

Or by watching other people

- Subway, Frito-Lite

- How do you learn Alt-Tab

(4)

Same w/ touch device

- learn in TV ad

UI must communicate by itself how to do it

Recognition vs Recall

↑
remembering w/
help

↑
most GUI
↑ can see what
commands you
can do

↑
just remember

↑
unix prompt
↑ low visibility

Good UI teaches

Twitter - some visible

- but need to remember hash tags and @

- or ~~have~~ even learn what they are for

MT = modified when retweeting

Can see how other people use Twitter

Reply button - pre fills @ username in tweet box

↑ "Self disclosure"

5

Direct manipulation

You are actual doing w/ objects
ie drag folder to trash

[hot File → Move to trash
or type "rm filename"]

Continuously viewable

Can see state of the system

Also scroll bars, image handles



Google Street View

drag around to pan

mouse wheel to zoom

but also buttons

drag + drop pushpin

double click to move down street

⑥

Shows businesses on map

Rotate menu

↳ w/ tool tips

Click not trivially reversible

Keyboard shortcuts

- must know about

Perceptual model

What are the pieces?

How do they work together

3 levels of models

- System model - how db set up
 - Interface model - how it looks
 - User model - how the user thinks it works
- ↳ what we build in 6.813

⑦ How we dial changed

human operator



dial tone + ###



+ Send

Underlying system changed

Copper circuit



packet switching



~~cell~~ (cell tones)

How we actually talk has not changed at all

Is electricity like water flowing?

Not technically

But fine to think about it

But is a thermostat like a valve or a switch

L2: Learnability

- no class on Monday
- HW1 (hall of fame & shame) out Mon, due next Sun
- start putting your project group together – use Groups Wanted page on wiki

UI Hall of Fame or Shame?



Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface 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 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

Spring 2012

6.813/6.831 User Interface Design and Implementation

3

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

mouse over

Spring 2012

6.813/6.831 User Interface Design and Implementation

4

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, we'll look at some other ways.

Today's Topics

- Learning approaches
- Interaction styles
- Conceptual models of UIs
- Consistency

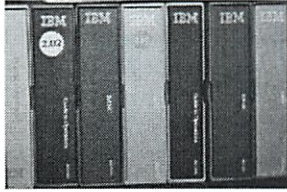
LEARNING APPROACHES

Spring 2012

6.813/6.831 User Interface Design and Implementation

8

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"

Spring 2012

6.813/6.831 User Interface Design and Implementation

9

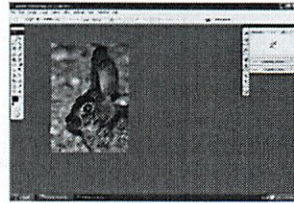
When computers first appeared in the world, there were some assumptions about how people would learn how to use the software. Programmers assumed that users would read the manual first – obviously not true. Companies assumed that their employees would take a class first – not always true. Even now that 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 already has a **goal** to achieve
 - “Get rid of the redeye from my photo.”
- User **explores** interface to find commands and features to satisfy the goal



Spring 2012

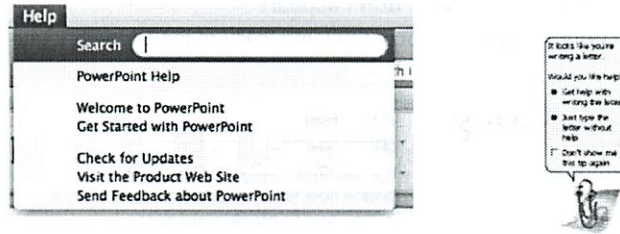
6.813/6.831 User Interface Design and Implementation

10

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



Spring 2012

6.813/6.831 User Interface Design and Implementation

11

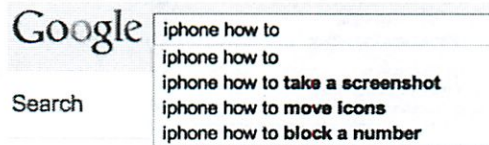
Only when they get stuck in their learning-by-doing will a typical user look for help. This affects the way help systems should be designed, because it means most users (even first-timers) are arriving at help with a goal already in mind and an obstacle they've encountered to achieving that goal. A help 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.

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?



Spring 2012

6.813/6.831 User Interface Design and Implementation

12

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?

Spring 2012

6.813/6.831 User Interface Design and Implementation

13

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, like pinch-zooming or Alt-Tab.

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.



INTERACTION STYLES

Spring 2012

6.813/6.831 User Interface Design and Implementation

14

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!

Spring 2012

6.813/6.831 User Interface Design and Implementation

15

It’s important to make a distinction between **recognition** (remembering with the help of a visible cue) and **recall** (remembering something with no help from the outside world). 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.

Command Language:

- User types in commands in an artificial language
 - all knowledge in the head; low learnability

`ls -l *.java`

Unix shell

`+6.831 site:mit.edu`

search engine query

`http://www.mit.edu/admissions/`

URL

Spring 2012

6.813/6.831 User Interface Design and Implementation

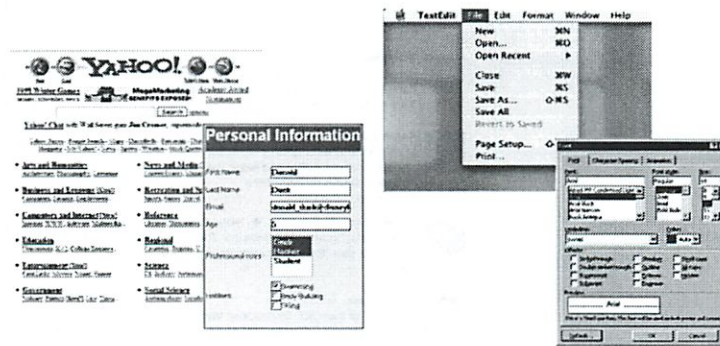
16

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.

Menus and Forms

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



Spring 2012

6.813/6.831 User Interface Design and Implementation

17

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).

Example: Twitter's Tweet-Creation UI

What aspects of this UI use knowledge in the head?

What aspects use knowledge in the world?



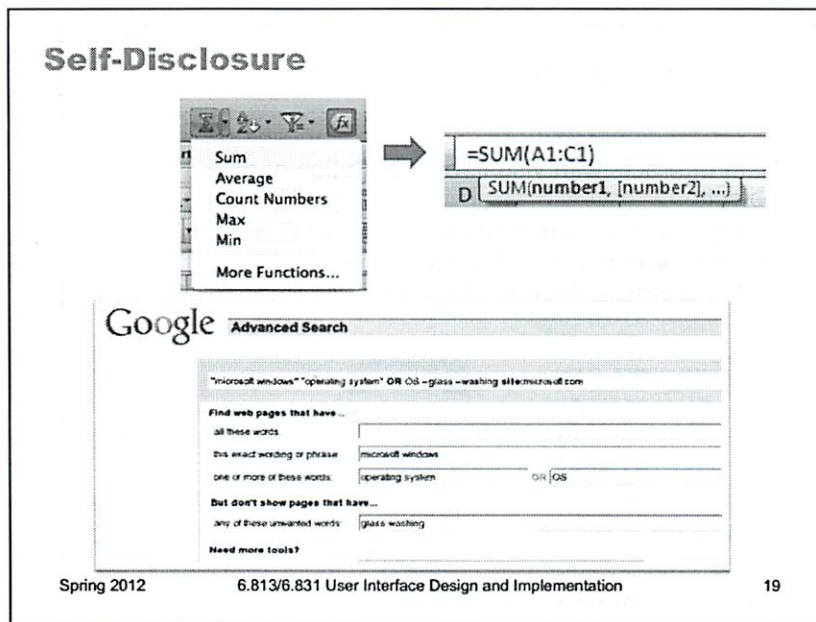
Spring 2012

6.813/6.831 User Interface Design and Implementation

18

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 by Geza Kovacs)

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

Spring 2012

6.813/6.831 User Interface Design and Implementation

20

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** (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.

Try It: Direct Manipulation

- What parts of Google Street View satisfy the definition of DM?
 - Continuous visual representation
 - Physical actions or labeled button presses
 - Rapid, incremental, reversible, immediately visible effects



Spring 2012

6.813/6.831 User Interface Design and Implementation

21

The Street View feature of Google Maps is a great example of direct manipulation in an interface. The interface allows the user to either click or click-drag to change their perspective and location in Google Maps. The interactive map gives the user a continuous visual representation of geographical data that responds rapidly, visually, and incrementally to physical mouse movements. (suggested by Feng Wu)

Go to:

http://maps.google.com/maps?q=77+Mass+Ave,+Cambridge,+Massachusetts+02139&layer=c&z=17&iwloc=A&sll=42.359057,-71.093571&cbp=13,117.0,0,0,0&cbll=42.359072,-71.093612&hl=en&ved=0CAoQ2wU&sa=X&ei=NxU1T83BGYzcNY_K8KIJ

shoutkey.com/beeive

CONCEPTUAL MODELS

Spring 2012

6.813/6.831 User Interface Design and Implementation

22

Models

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



Spring 2012

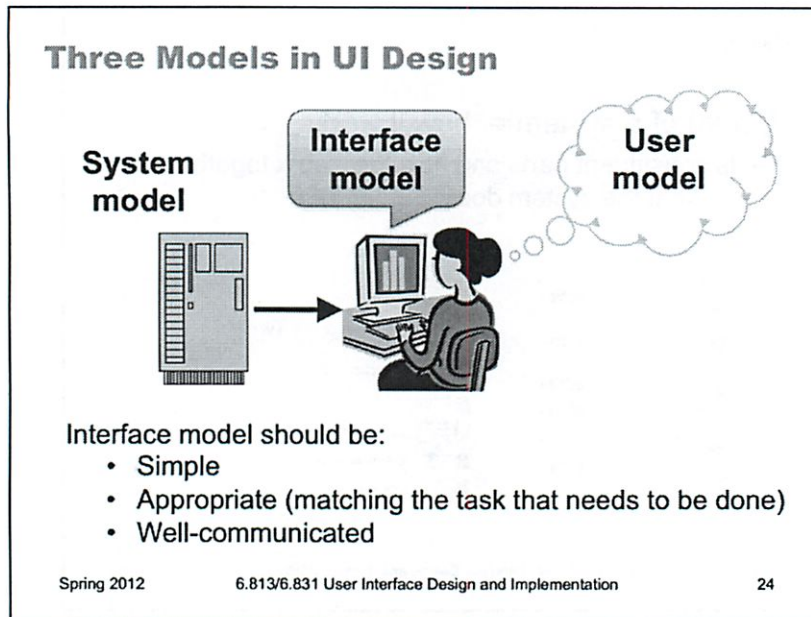
6.813/6.831 User Interface Design and Implementation

23

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.

Example: Telephone

- Conversation model has not changed
 - despite drastic changes in system model (copper circuit → packet switching → cells)



- Interface model for dialing has evolved a lot
 - human operator → dialtone+### → ###+Send

Spring 2012

6.813/6.831 User Interface Design and Implementation

25

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.

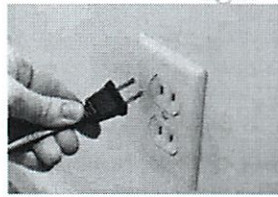
Similarly, a cell phone presents the same simple interface model as a conventional wired phone, even though its system model is quite a bit more complex. A cell phone conversation may be handed off from one cell tower to another as the user moves around. This detail of the system model is hidden from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

Example: Electricity as Water

System model

- AC current
- circuit
- electrons don't transfer



Interface model

- power flowing like water

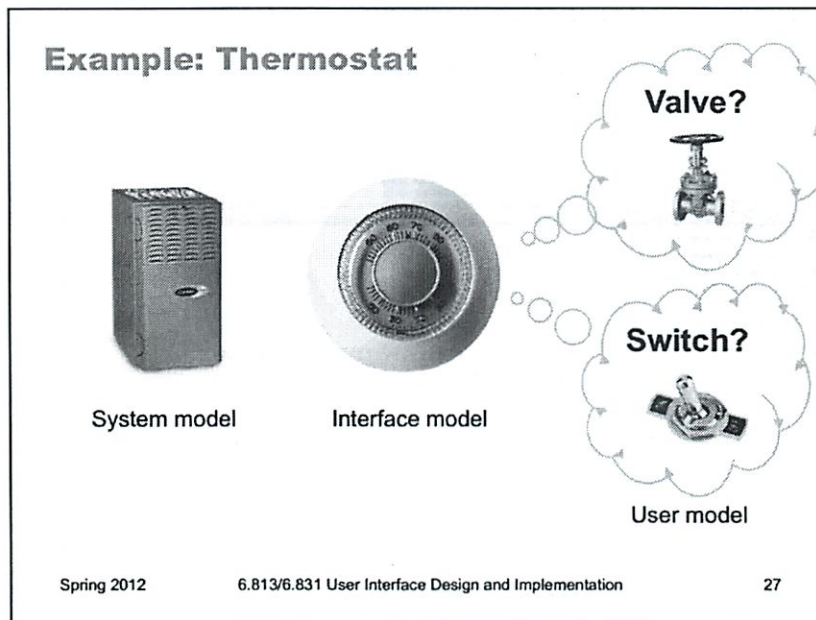


Spring 2012

6.813/6.831 User Interface Design and Implementation

26

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.

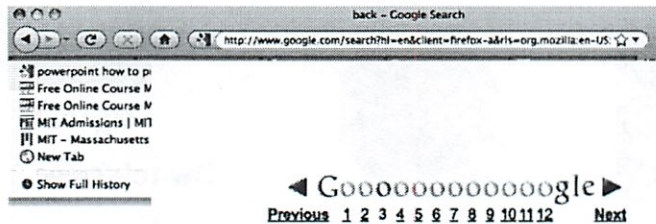


But a wrong user model can lead to problems, as well. Consider a household thermostat, which controls the temperature of a room. If the room is too cold, what's the fastest way to bring it up to the desired temperature? Some people would say the room will heat faster if the thermostat is turned all the 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.

Example: Back vs. Previous



Spring 2012

6.813/6.831 User Interface Design and Implementation

28

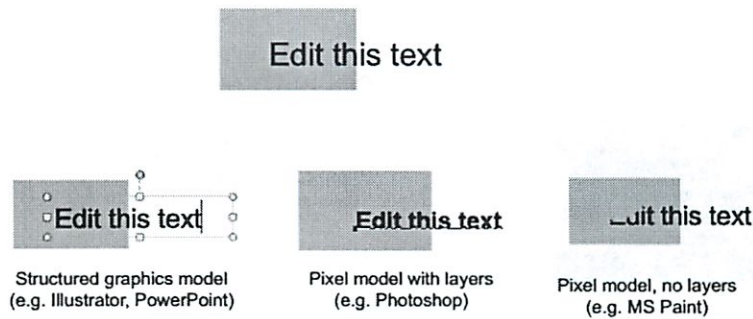
Here's an example drawn directly from graphical user interfaces: the Back button in a web browser. What is the model for the behavior of Back? Specifically: how does the *user* think it behaves (the mental model), and how does it *actually* behave (the system model)?

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>

Example: Graphical Editing



Spring 2012

6.813/6.831 User Interface Design and Implementation

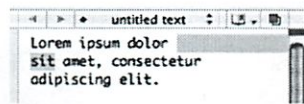
29

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

Spring 2012

6.813/6.831 User Interface Design and Implementation

30

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

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?





CONSISTENCY

Spring 2012

6.813/6.831 User Interface Design and Implementation

32

Consistency

- Similar things should look and act the same
- Different things should look different
 - also called the principle of “least surprise”
- Consistency allows the user to transfer their existing knowledge easily to a new UI

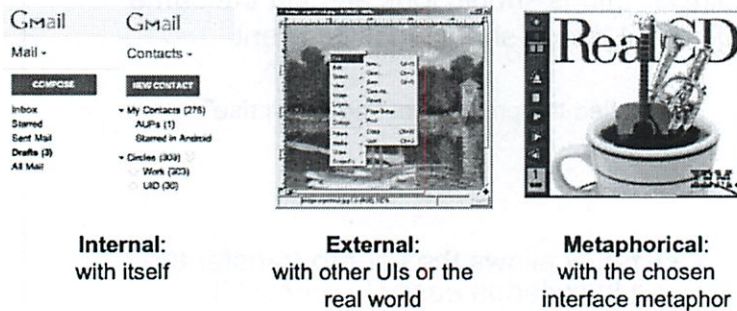
Spring 2012

6.813/6.831 User Interface Design and Implementation

33

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.

Kinds of Consistency



Spring 2012

6.813/6.831 User Interface Design and Implementation

34

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

- Advantages
 - Highly learnable when appropriate
 - Hooks into user's existing mental models very easily
- Dangers
 - Often hard for designers to find
 - May be deceptive
 - May be constraining
 - Metaphor always breaks down



Desktop metaphor



Trashcan metaphor



Typewriter metaphor

Spring 2012

6.813/6.831 User Interface Design and Implementation

35

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?)

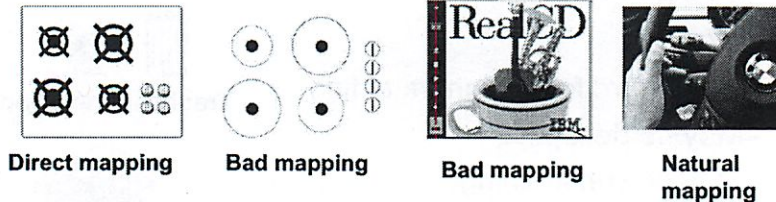
The basic rule for metaphors is: use it if you have one, but don't stretch for one if you don't. Appropriate metaphors can be very hard to find, particularly with real-world objects. The designers of RealCD stretched hard to use their CD-case metaphor (since in the real world, CD cases don't even play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

Natural Mapping: Consistency of Layout

- When possible, the physical arrangement of controls should match arrangement of function
- Best mapping is **direct**, but natural mappings don't have to be direct if they have an easy mental model



Spring 2012

6.813/6.831 User Interface Design and Implementation

36

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

Internal Consistency in Wording

Course VI Underground Guide Evaluations

[Home](#) [Search](#) [Teacher](#)

Published UG reviews
[Browse](#) or [Search](#) through past published evaluations

Underground Guide Review
Lecturer's Comments
Please enter your lecturer's comments below. These comments will be used to answer any or all of the following questions:

Browse published evaluations
[or visit our search page](#)
Fall 2007 evaluations will be available 2008 02 28

Preview/Review:
Preview your class's evaluation [HERE](#)
[Read Student Evaluations](#) - Read what students had about the class.
[Read Underground Guide Review](#) - Read the Underground Guide review for

Spring 2012 6.813/6.831 User Interface Design and Implementation 37

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: Speak the User's Language

- Use common words, not techie jargon
 - But use domain-specific terms where appropriate
- Allow aliases/synonyms in command languages



Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface Design and Implementation

38

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.

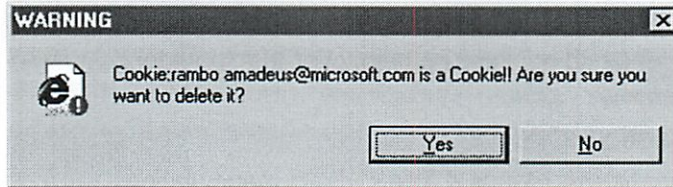
When designing an interface that requires the user to type in commands or search keywords, support as many aliases or synonyms as you can. Different users rarely agree on the same name for an object or command. One study found that the probability that two users would mention the same name was only 7-18%. (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

- Learning approaches
 - learn by doing, seeking help, over-the-shoulder learning
- Interaction styles
 - commands, menus & forms, direct manipulation
- Conceptual models
 - system vs. interface vs. user models
- Consistency
 - internal, external, metaphorical

Next Time: UI Hall of Fame or Shame?



Source: Interface Hall of Shame

C3 Learnability

2/15

(no class mon)

HW1 out, due Sun

Form groups, 1st milestone soon (GA1)

- initial analysis
- pick problem / project

Hall of Fame / Shuno

- MOMA online exhibition
- starts up w/ Tech requirements
- a grid of boxes
- rolling over one brings up a sticky notes
- metaphor: wall of past it notes
- but it doesn't feel like it
- ~~the~~ museum might want people to get lost so they serendipitous discovery
- very simple interface

(2)

Quiz

Direct manipulation

- the scroll bar itself is a direct manipulation of the moving x, y pos (low level thing)
- larger task is not direct manipulation

(much clearer now)

(only writing what I missed)

Self disclosure - move from know. in world \rightarrow know. in head
Not really how system is exposed to user

He hates videos - can't skip
Other users like it

Model of a system \rightarrow how it works

- how the parts work together

UI should communicate this model of how the system works

③

Example: Back vs Previous

Back button - last page/URL that is in browser history
in browser

Previous button on Google - goes to previous content

Sometimes they go to the same place

But not always

- if you go to pg 10
- then hit previous → at 9
- then
 - "Back" goes to 10
 - "previous" goes to 8

Fixing it would be even more complex

Object Graphical Editing

- structured like Powerpoint, Illustrator
- pixel made in layers like I made in Photoshop
- Straight pixel model like MS Paint

Attendances: bounding box

Text cursor 

9

Text editor - has line breaks auto (word wrap)

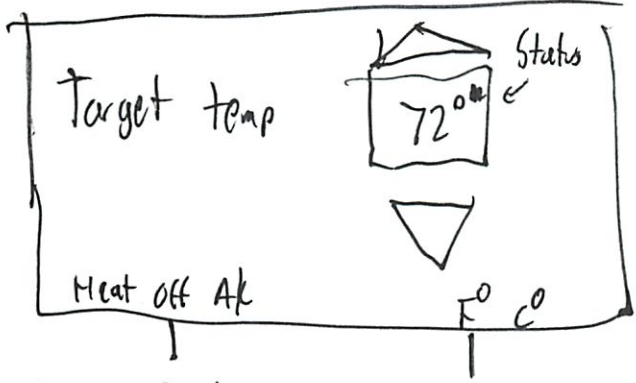
Enter + Space very different in word processing
than type writer

Underlining on type writer: typing underscores over item!

Thermostat is like a valve:

- round
- radiators are actually valves
- but many are just switches


Can you design better?



← narrows not colored
red and blue
instead uniform

? Switch
on bottom

Other ops

On  ← handle
off

Similar to mixer
but w/ heat on/off indicator

? I would avoid
actually prob ok

6

System that knows outside temp

- controls heat / cool with that
- or house fan / vents

Other groups

- time dial
(don't fully understand)

- linear control
00000000 current indicator lights
└──────────┘
^
target

- heat on indicator light



So know that only 2 states; on or off
could users think light brightness matters?

people could think its an abstraction

- no degree display of how much it is on?

- is degrees even good

└──────────┘
unhappy good

⑥

- Have a 'just right' button
 - Or on/off/hold
 - heat rate slider
 - not hooked up
 - bring to user
 - pretending system model is not what it is
 - Progress bar until hit desired temp
 - but won't communicate rate is not changing
 - Phone booth testing env
 - instant preview
-

Sinks - immediate feedback

New product: Nest thermostat

- machine learning thermostat
- it learns your pattern of activities
- tells you how long to set point

⑦

How would you test it?
See if people make errors

Need to set up scenarios just right

Consistency

(30 slides for 5 min)

Easy way to get learnability: copy others

Low surprise

Don't do too much.

1. Be consistent w/ itself

2. Be " w/ other UIs or real world

3. " " w/ chosen metaphors (if you use one...)

Metaphors

Can be highly learnable

but will break down at some point - always

may be constraining

8

Mapping Controls

Stove



↑ burners

good
↔



↑ Controls



bad!

↔



~~bad!~~

Two signals stalk

- does not directly map
- but feels natural
- since w/ wheel

Wording

Use common terms

evaluation = reviews

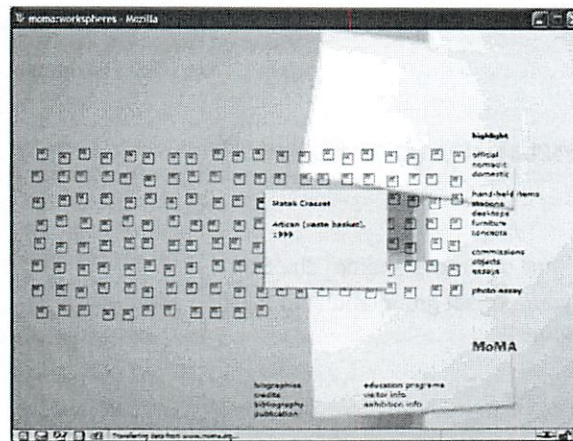
Prof ~~like~~ use short words

in your writing too

L3: Learnability (continued)

- HW1 (hall of fame & shame) due Sun
- form your project group and start thinking about your project

UI Hall of Fame or Shame?



Suggested by Vishy Venugopalan

Spring 2011

6.813/6.831 User Interface Design and Implementation

2

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

Spring 2012

6.813/6.831 User Interface Design and Implementation

6

Models

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



Spring 2012

6.813/6.831 User Interface Design and Implementation

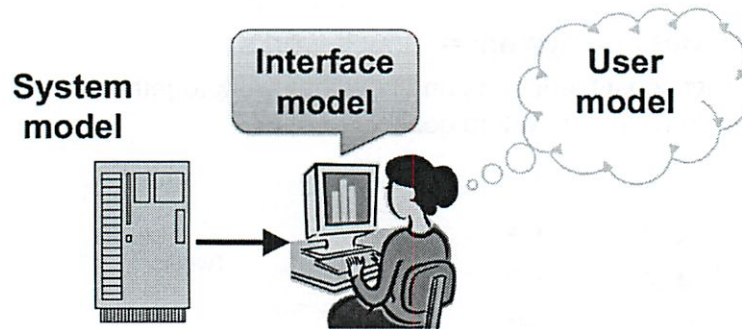
7

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.

Three Models in UI Design



Interface model should be:

- Simple
- Appropriate (matching the task that needs to be done)
- Well-communicated

Spring 2012

6.813/6.831 User Interface Design and Implementation

8

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.

Example: Back vs. Previous



Spring 2012

6.813/6.831 User Interface Design and Implementation

9

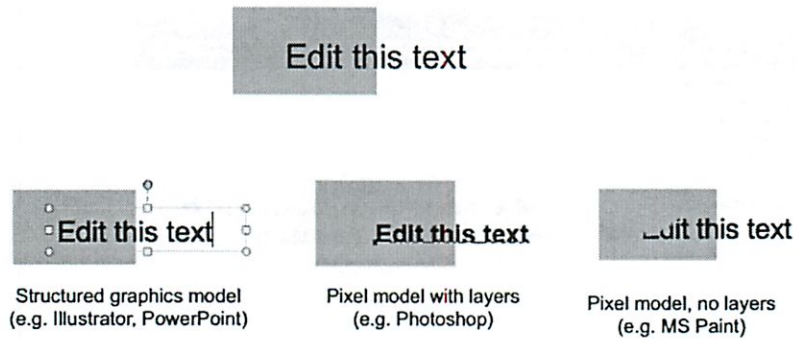
Here's an example drawn directly from graphical user interfaces: the Back button in a web browser. What is the model for the behavior of Back? Specifically: how does the *user* think it behaves (the mental model), and how does it *actually* behave (the system model)?

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>

Example: Graphical Editing



Spring 2012

6.813/6.831 User Interface Design and Implementation

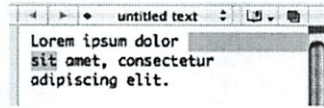
10

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

Spring 2012

6.813/6.831 User Interface Design and Implementation

11

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

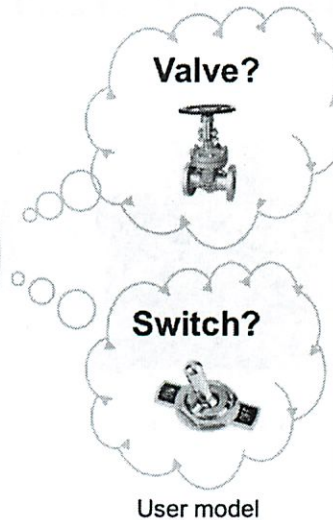
Example: Thermostat



System model



Interface model



User model

Spring 2012

6.813/6.831 User Interface Design and Implementation

12

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?



One Possible Design



Spring 2012

6.813/6.831 User Interface Design and Implementation

14

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

Spring 2012

6.813/6.831 User Interface Design and Implementation

15

Consistency

- Similar things should look and act the same
- Different things should look different
 - also called the principle of “least surprise”
- Consistency allows the user to transfer their existing knowledge easily to a new UI

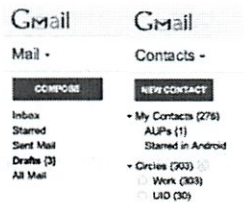
Spring 2012

6.813/6.831 User Interface Design and Implementation

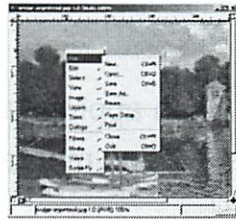
16

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.

Kinds of Consistency



Internal:
with itself



External:
with other UIs or the
real world



Metaphorical:
with the chosen
interface metaphor

Spring 2012

6.813/6.831 User Interface Design and Implementation

17

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

- Advantages
 - Highly learnable when appropriate
 - Hooks into user's existing mental models very easily
- Dangers
 - Often hard for designers to find
 - May be deceptive
 - May be constraining
 - Metaphor always breaks down



Desktop metaphor



Trashcan metaphor



Typewriter metaphor

Spring 2012

6.813/6.831 User Interface Design and Implementation

18

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?

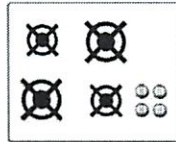
Metaphors can also be constraining. Strict adherence to the desktop metaphor wouldn't scale, because documents would always be full-size like they are in the real world, and folders wouldn't be able to have arbitrarily deep nesting.

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 *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.

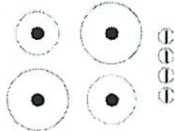
Most of all, using a metaphor doesn't save an interface that does a bad job communicating itself to the user. Although RealCD's model was metaphorical -- it opened like a CD case, and it had a liner notes booklet inside the cover -- these features had such poor visibility and perceived affordances that they were ineffective.

Natural Mapping: Consistency of Layout

- When possible, the physical arrangement of controls should match arrangement of function
- Best mapping is **direct**, but natural mappings don't have to be direct if they have an easy mental model



Direct mapping



Bad mapping



Bad mapping



Natural mapping

Spring 2012

6.813/6.831 User Interface Design and Implementation

19

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?

Internal Consistency in Wording

Course VI Underground Guide Evaluations

[Home](#) [Search](#) [Teacher](#)

Published UG reviews
[Browse](#) or [Search](#) through past published evaluations

Underground Guide Review
Lecturer's Comments
Please enter your lecturer's comments below. These comments will be used to answer any or all of the following questions

Browse published evaluations
or visit our [search page](#)
Fall 2007 evaluations will be available 2008-02-28

Preview/Review:
Preview your class's evaluation [HERE](#)
[Read Student Evaluations](#) - Read what students said about the class.
[Read Underground Guide Review](#) - Read the Underground Guide review for

Spring 20126.813/6.831 User Interface Design and Implementation20

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: Speak the User's Language

- Use common words, not techie jargon
 - But use domain-specific terms where appropriate
- Allow aliases/synonyms in command languages



Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface Design and Implementation

21

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.

AFFORDANCES

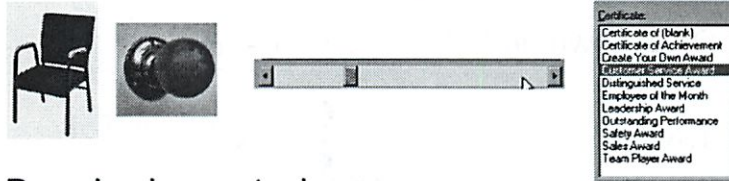
Spring 2012

6.813/6.831 User Interface Design and Implementation

22

Affordances

- Perceived and actual properties of a thing that determine how the thing could be used



- Perceived vs. actual



Spring 2011

6.813/6.831 User Interface Design and Implementation

23

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.

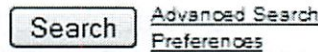
Note that **perceived** affordance is not the same as **actual** affordance. A facsimile of a chair made of papier-mache has a perceived affordance for sitting, but it doesn’t actually afford sitting: it collapses under your weight. Conversely, a fire hydrant has no perceived affordance for sitting,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

Recall the textbox from our first lecture, 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 on the right, 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.

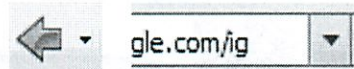
The original definition of affordance (from psychology) referred only to actual properties, but when it was imported into human computer interaction, perceived properties became important too. Actual ability without any perceivable ability is an undesirable situation. We wouldn’t call that an affordance. 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 room allow you to exit, if you know how.

Use Appropriate Affordances

- Buttons & links



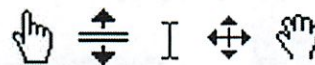
- Drop-down arrows



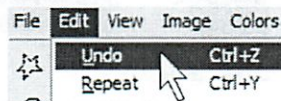
- Texture



- Mouse cursor



- Highlight on mouseover



Spring 2011

6.813/6.831 User Interface Design and Implementation

24

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.

Texture suggests that something can be clicked and dragged – relying on the physical metaphor, that physical switches and handles often have a ridged or bumpy surface for fingers to more easily grasp or push.

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 of the mouse, it suggests that you can interact with it by clicking.

What Can You Do With This Page?



Suggested by Dina Betser

Spring 2011

6.813/6.831 User Interface Design and Implementation

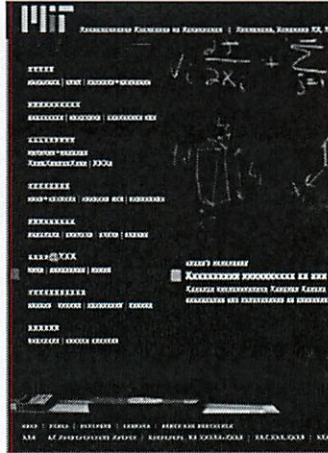
25

If the user is trying to view all the events at once on the CPW website, the user may end up clicking through all the days individually. It turns out that the graphic in the center page is actually a link to a nifty search interface that lets the user look at all the event listings in addition to other cool functionalities, but the graphic doesn't have strong affordances for interaction. It's mostly a big logo, so what does a typical user do? Glance at it and then ignore it, scanning the page instead for things that look like actions, such as the clearly marked hyperlinks at the bottom. The "click here to search" text in the logo **doesn't work**.

It is very easy to miss the search interface altogether because of the poor visibility of the search feature and the lack of affordances that the graphic is a link. (example and explanation due to Dina Betser)

Try It: Playing with Affordances

- Use Javascript to obscure all the text on a page
- Visit several pages, e.g.:
 - MIT's home page
 - 6.813/6.831 home page
- What do the affordances tell you nonverbally?
- Where do the affordances lie?



Spring 2012

6.813/6.831 User Interface Design and Implementation

26

Here's the Javascript code:

```
var result = document.evaluate("//text()", document.body, null,
XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE, null);for (var i = 0; i < result.snapshotLength; ++i) {var
node = result.snapshotItem(i);if ((node.textContent+ "").match(/\w/)&&node.parentNode.nodeName !=
"STYLE") {node.textContent = node.textContent.replace(/[A-Z0-9]/g, "X").replace(/[a-z]/g, "x");}}void
0
```

One way to use it is to open your browser's Javascript console and just paste the code in; it will change the current page. Another way to use it is to create a new bookmark in your browser, and use as the URL javascript: followed by the code given above. Clicking on this bookmark will run the Javascript on the current page. (This is called a *bookmarklet*, and it's an one way to modify web pages you don't own.)

FEEDBACK

Spring 2012

6.813/6.831 User Interface Design and Implementation

27

Feedback:
Actions Should Have Immediately Visible Effects

- Low-level feedback

- e.g. push button



- High-level feedback

- model state changes
- new web page starts loading

The third and final aspect of visibility is **feedback**: how the system changes when you perform an action.

When the user invokes a part of the interface, it should appear to respond. Push buttons should depress and release. Scrollbar thumbs and dragged objects should move with the mouse cursor. Pressing a key should make a character appear in a textbox.

Low-level feedback is provided by a view object itself, like push-button feedback. This kind of feedback shows that the interface at least took notice of the user's input, and is responding to it. (It also distinguishes between disabled widgets, which don't respond at all.)

High-level feedback is the actual result of the user's action, like changing the state of the model.

Perceptual Fusion

- Two stimuli within the same perceptual cycle ($T_p \sim 100\text{ms}$ [50-200 ms]) appear **fused**
- Consequences
 - $1/T_p$ frames/sec is enough to perceive a moving picture (10 fps OK, 20 fps smooth)
 - Computer response $< T_p$ feels instantaneous
 - Causality is strongly influenced by fusion

Spring 2011

6.813/6.831 User Interface Design and Implementation

29

One interesting effect of human perceptual system is **perceptual fusion**. Here's an intuition for how fusion works. Our "perceptual processor" runs at a certain frame rate, grabbing one frame (or picture) every cycle, where each cycle takes T_p seconds. Two events occurring less than the cycle time apart are likely to appear in the same frame. If the events are similar – e.g., Mickey Mouse appearing in one position, and then a short time later in another position – then the events tend to *fuse* into a single perceived event – a single Mickey Mouse, in motion.

The cycle time of the perceptual processor can be derived from a variety of psychological experiments over decades of research (summarized in Card, Moran, Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, 1983). 100 milliseconds is a typical value which is useful for a rule of thumb. But it can range from 50 ms to 200 ms, depending on the individual (some people are faster than others) and on the stimulus (for example, brighter stimuli are easier to perceive, so the processor runs faster).

Perceptual fusion is responsible for the way we perceive a sequence of movie frames as a moving picture, so the parameters of the perceptual processor give us a lower bound on the frame rate for believable animation.

10 frames per second is good enough for a typical case, but 20 frames per second is better for most users and most conditions.

Perceptual fusion also gives an upper bound on good computer response time. If a computer responds to a user's action within T_p time, its response feels instantaneous with the action itself. Systems with that kind of response time tend to feel like extensions of the user's body. If you used a text editor that took longer than T_p response time to display each keystroke, you would notice.

Fusion also strongly affects our perception of causality. If one event is closely followed by another – e.g., pressing a key and seeing a change in the screen – and the interval separating the events is less than T_p , then we are more inclined to believe that the first event caused the second.

Response Time

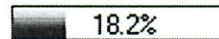
< 0.1 s: seems instantaneous

0.1-1 s: user notices the delay

1-5 s: display busy indicator



> 1-5 s: display progress bar



Spring 2011

6.813/6.831 User Interface Design and Implementation

30

Perceptual fusion provides us with some rules of thumb for responsive feedback.

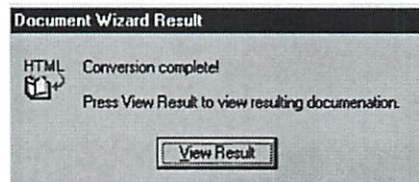
If the system can perform a command in less than 100 milliseconds, then it will seem instantaneous, or near enough. As long as the result of the command itself is clearly visible – e.g., in the user's locus of attention – then no additional feedback is required.

If it takes longer than the perceptual fusion interval, then the user will notice the delay – it won't seem instantaneous anymore. *Something* should change, visibly, within 100 ms, or perceptual fusion will be disrupted. Normally, however, ordinary low-level feedback is enough to satisfy this requirement, such as a push-button popping back, or a menu disappearing.

One second is a typical turn-taking delay in human conversation – the maximum comfortable pause before you feel the need to fill the gap with something, even if it's just "uh" or "um". If the system's response will take longer than a second, then it should display additional feedback. For short delays, the hourglass cursor (or spinning cursor, or throbber icon shown here) is a common design pattern. For longer delays, show a progress bar, and give the user the ability to cancel the command.

Note that progress bars don't necessarily have to be *accurate*. (This one is actually preposterous – who cares about 3 significant figures of progress?) An effective progress bar has to show that progress is being made, and allow the user to estimate completion time at least within an order of magnitude – a minute? 10 minutes? an hour? a day?

Useless Feedback vs. Useful Feedback

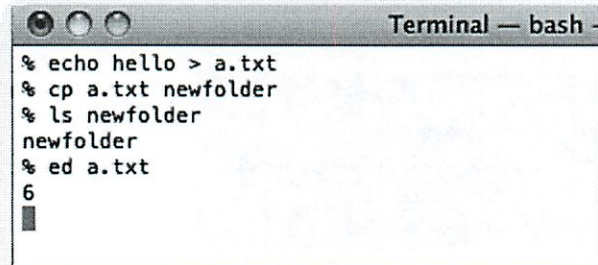


Source: Interface Hall of Shame

Feedback is important, but don't overdo it. This dialog box demands a click from the user. Why? Does the interface need a pat on the back for finishing the conversion? It would be better to just skip on and show the resulting documentation.

Try It: Unix shell

- Open a command prompt and run:



```
Terminal — bash -
% echo hello > a.txt
% cp a.txt newfolder
% ls newfolder
newfolder
% ed a.txt
6

```

- Think about:
 - affordances
 - feedback

Spring 2011

6.813/6.831 User Interface Design and Implementation

32

Let's summarize with a case study of weak learnability: the Unix command line. Unix may be beautiful for many reasons, but learnability is not one of them.

The **actions** available to the user are completely invisible; the user must recall a command name from memory, along with the syntax for its arguments.

The **state** of the underlying system is likewise mostly hidden. Many users customize their prompts to make some state visible, such as the current directory or the hostname. The contents of the current directory are *not* visible, even though many commands operate on files.

The **feedback** from a command is minimal – in fact, one Unix design principle is that commands should say *nothing* when they succeed. But that's not a good thing. It's true that a generic feedback message like "command completed successfully" would indeed be useless; the subsequent appearance of a command prompt is sufficient feedback that the command has completed. So what kind of visible feedback *would* be useful?

INFORMATION SCENT

Spring 2012

6.813/6.831 User Interface Design and Implementation

33

Information Scent

- Information foraging theory
 - Humans gathering information can be modeled like animals gathering food
 - Constantly evaluating and making decisions to maximize information collected against cost of obtaining it
- Information scent
 - Cues on a link that indicate how profitable it will be to follow the link to its destination

Spring 2011

6.813/6.831 User Interface Design and Implementation

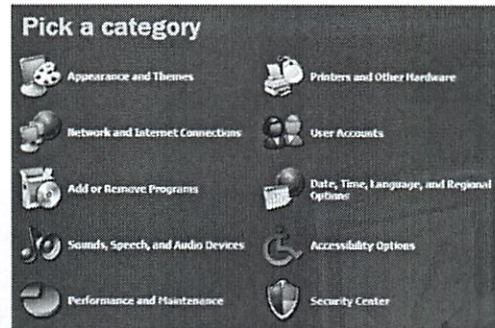
34

Users depend on visible cues to figure out how to achieve their goals with the least effort. For information gathering tasks, like searching for information on the web, it turns out that this behavior can be modeled much like animals foraging for food. An animal feeding in a natural environment asks questions like: Where should I feed? What should I try to eat (the big rabbit that's hard to catch, or the little rabbit that's less filling)? Has this location been exhausted of food that's easy to obtain, and should I try to move on to a more profitable location? **Information foraging theory** claims that we ask similar questions when we're collecting information: Where should I search? Which articles or paragraphs are worth reading? Have I exhausted this source, should I move on to the next search result or a different search? (Pirolli & Card, "Information Foraging in Information Access Environments," *CHI '95*.)

An important part of information foraging is the decision about whether a hyperlink is worth following – i.e., does this smell good enough to eat? Users make this decision with relatively little information – sometimes only the words in the hyperlink itself, sometimes with some context around it (e.g., a Google search result also includes a snippet of text from the page, the site's domain name, the length of the page, etc.) These cues are **information scent** – the visible properties of a link that indicate how profitable it will be to follow the link. (Chi et al, "Using Information Scent to Model User Information Needs and Actions on the Web", *CHI 2001*.)

Give Good Information Scent

- A link should smell like the content it leads to



Spring 2011

6.813/6.831 User Interface Design and Implementation

35

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.

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.

Learnability
Efficiency
Errors
Simplicity

Good & Bad Information Scent

- To learn more about this site, [click here](#)
- Learn more about this site [here](#)
- Learn more about this site
- Link to this site's [about page](#).

- [Learn more about this site](#)
- [About](#)

Audio and TV
Books
Computing
Fashion
Furniture
Gardening

Audio and TV: Camcorders, DVD and Video, Hi-Fi...
Books: Bestsellers, Factual, Education...
Computing: Computers, Games, Printers
Fashion: Mens, Womens, Kids...
Furniture: Bathrooms, Bedrooms, Kitchen...
Gardening: Seeds, Plants, Pots

IAP 2012
6.470 IAP Web Programming Competition

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.

Lots of scent but hard to scan

RENT MONKEY

Home | Search Listings | Manage Listings and Profiles | Residence History | Browse Residences

Search listings, by MIT students, for MIT students
Many of these listings are unofficial, but they can help guide you towards places with upcoming vacancies. Other listings may be posted by the MIT Off-campus Housing Office.

Advertise an off-campus vacancy or sublet
Want to announce a vacancy or a summer sublet? Manage the listings and residence profiles you've edited on this site. For on-campus lottery and sublets, please visit the [graduate housing website](#).

See what others have said about a residence
Check if other MIT students have written about a particular residence. Look at the rent history of a residence to see how much you should be paying.

Browse where other students are living
Look at where other MIT students are living to guide where you may want to live.

© 2008 GSC HCA, MIT Housing, Robert Wang

Website
Feedback
Disclaimer
MIT Rental
Guide
GeoSapla
Search
Ask MIT
Housing
About

More resources:
GSC HCA | MIT Housing

Spring 2011 6.813/6.831 User Interface Design and Implementation 37

<http://rentmonkey.mit.edu/account/home>

Hierarchy of Exploration

- Quick glance
 - short salient words & icons
- Closer look
 - description, lists of keywords
- Probing with the mouse
 - cursor change, highlight, tooltip
- Clicking through
- Trying out

Tooltips are another nice design pattern for providing a more descriptive label of a small control, and also a place for making other shortcuts visible.

Summary

- Conceptual models
 - system vs. interface vs. user models
- Consistency
 - internal, external, metaphorical
- Affordances
- Feedback
- Information scent

VI hall of shame: Ask ya to ~~confirm~~ confirm each file individually
no "Yes to All"

bad efficiency

but Yes to All has safety problems

Could display whole list of choices in a
scroll box w/ checkmarks

W. Nassqvist User models - about what users have in their head
(got wrong)
not types of user profiles

Natural mapping - physical arrangement of controls
is consistent w/ ~~physical~~ operations
(The stove example - ~~all~~ think back)

Visibility

History: Human Factors - minimizing mistakes
- air force, etc

Data Entry - efficiency important

② Now it's about new buyers

Learnability (cont.)

Affordances - how we learn to use something

Set of properties of object that determine how the thing can be used

Chair

looks sturdy

flat surface

raised to good level for sitting

resembles a chair

knob

affords grasping

round - good to turn

affords pulling

door handle

flat or handle



We can easily see how door opens

②

interface should not lie - should not contradict itself

should be consistent w/ other things that are similar

- like many chairs look similar
- are cultural
- care about internationalization

Ridge pts on scrollbar - just so you think you can drag

Buttons looking 3D - from real life

Hyperlinks - totally invented

- blue is a horrible color
- Underline makes things hard to read
- designers have been trying to get rid of it

Mouse Cursors - change based on what you hover over

Highlight on mouse over

Large textboxes - suggests multiple addresses

(3)

MIT CPW page - bad affordances

He's built an XXX bookmarklet that replaces text w/ xxx

You won't notice the ~~drop down box~~ expand box in Stellar
- so it starts open

Immediately visible feedback

- a button depressed state

- Needs to be < 100 ms to allow us to notice
- feels like feedback

Response time

< 100 ms - seems instantaneous

1 - 1 sec - user notices - hour glass

1 - 5 sec - spin wait

> 5 sec - progress bar

Don't display a "I'm done" dialog box
if it's clear you are done

exception: DVD burning - can't notice it's done easily

9

Unix shell

'if everything ok → say nothing

↳ good not to pop up all the time
but its not clear it worked

ed - old fashioned editor for when you were
using text printer

You need to ask it what is in the file

Need to have the model in your head the whole time

or you can tab to autocomplete

- efficiency

- visibility ← not really in Unix

Information Scent

leave traces in UI so users can find way around
from foraging ~~all~~ of animals looking for food

We are constantly evaluating a website - will it answer
my question

ie: Printers and Other Hardware

- hardware includes printer

- but people commonly use Control Panel for printers

5

Don't call your links click here, more

- Since it users scanning page - sees underlines

Can think of hierarchy of costs that user will do

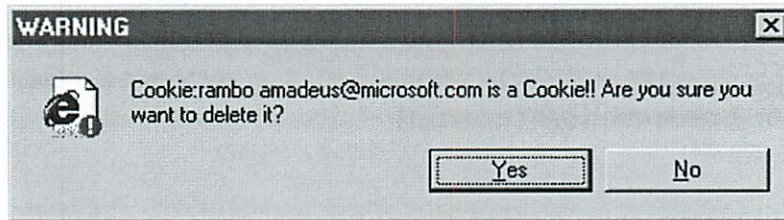
Chunking

next time

L4: Learnability (cont'd)

- HW1 (hall of fame & shame) due Sun
- GR1 (project proposal) out Mon, due next Sun

UI Hall of Fame or Shame?



Source: Interface Hall of Shame

Spring 2012

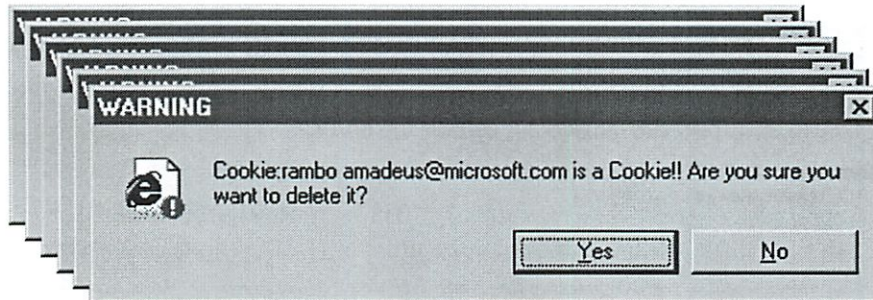
6.813/6.831 User Interface Design and Implementation

2

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).

Put aside the fact that the message is almost tautological ("Cookie... is a Cookie") and overexcited ("!!"). Does it give the user enough information to make a decision?

Hall of Shame



Source: Interface Hall of Shame

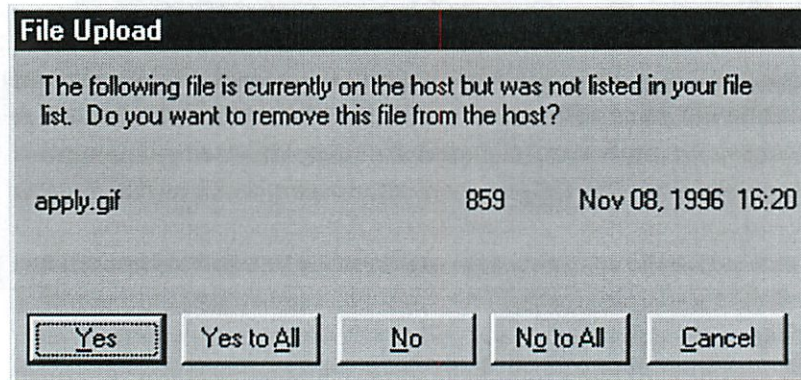
Spring 2012

6.813/6.831 User Interface Design and Implementation

3

Suppose you selected all your cookie files and tried to delete them all in one go. You get one dialog for every cookie you tried to delete! What button is missing from this dialog?

Hall of Fame or Shame?



Source: Interface Hall of Shame

Spring 2012

6.813/6.831 User Interface Design and Implementation

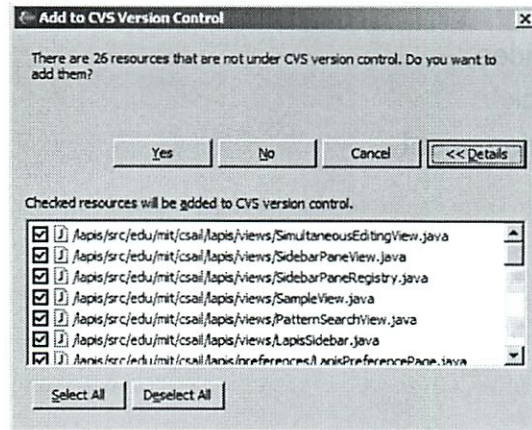
4

One way to fix the too-many-questions problem is Yes To All and No To All buttons, which short-circuit the rest of the questions by giving a blanket answer. That's a helpful shortcut, which improves **efficiency**, but this example shows that it's not a panacea.

This dialog is from Microsoft's Web Publishing Wizard, which uploads local files to a remote web site. Since the usual mode of operation in web publishing is to develop a complete copy of the web site locally, and then upload it to the web server all at once, the wizard suggests deleting files on the host that don't appear in the local files, since they may be orphans in the new version of the web site.

But what if you know there's a file on the host that you **don't** want to delete? What would you have to do?

Hall of Fame



Spring 2012

6.813/6.831 User Interface Design and Implementation

5

If your interface has a potentially large number of related questions to ask the user, it's much better to aggregate them into a single dialog. Provide a list of the files, and ask the user to select which ones should be deleted. Select All and Unselect All buttons would serve the role of Yes to All and No to All.

Here's an example of how to do it right, found in Eclipse. If there's anything to criticize in Eclipse's dialog box, it might be the fact that it initially doesn't show the filenames, just their count --- you have to press Details to see the whole dialog box. Simply knowing the *number* of files not under CVS control is rarely enough information to decide whether you want to say yes or no, so most users are likely to press Details anyway.

Today's Topics

- Affordances
- Feedback
- Information scent

AFFORDANCES

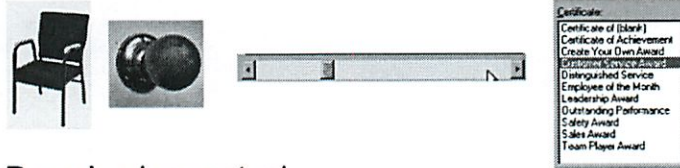
Spring 2012

6.813/6.831 User Interface Design and Implementation

9

Affordances

- Perceived and actual properties of a thing that determine how the thing could be used



- Perceived vs. actual



Spring 2012

6.813/6.831 User Interface Design and Implementation

10

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.

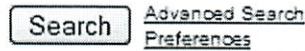
Note that **perceived** affordance is not the same as **actual** affordance. A facsimile of a chair made of papier-mache has a perceived affordance for sitting, but it doesn’t actually afford sitting: it collapses under your weight. Conversely, a fire hydrant has no perceived affordance for sitting,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

Recall the textbox from our first lecture, 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 on the right, 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 room allow you to exit, if you know how.

Use Appropriate Affordances

- Buttons & links



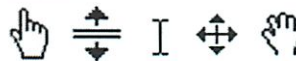
- Drop-down arrows



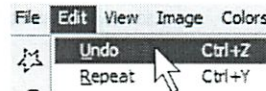
- Texture



- Mouse cursor



- Highlight on mouseover



Spring 2012

6.813/6.831 User Interface Design and Implementation

11

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.

Texture suggests that something can be clicked and dragged – relying on the physical metaphor, that physical switches and handles often have a ridged or bumpy surface for fingers to more easily grasp or push.

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 of the mouse, it suggests that you can interact with it by clicking.

Find the Affordances

SEND  Send Later **Saved** **Discard** Draft autosaved at 10:53 PM (

From:







To:






Add Cc Add Bcc

Subject:



Attach a file Insert: Invitation Canned responses ▾

☐ Boomerang this message if I don't hear back ▾ In 2 days ▾

B **I** **U** **T** **↶T** **A** **T**       Check Spelling ▾

     < Plain Text

Add Cc

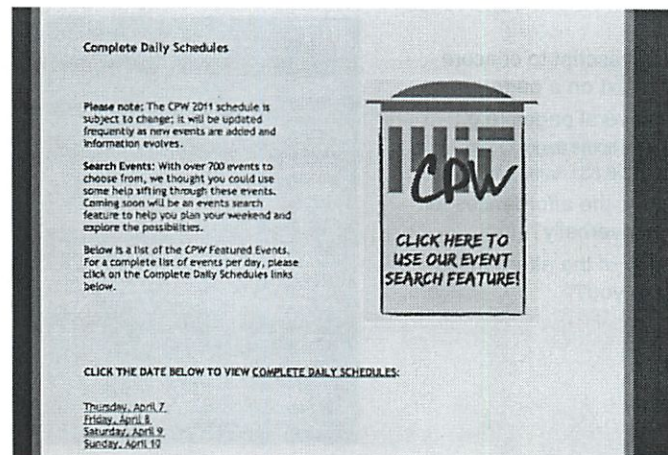
 

Spring 2012

6.813/6.831 User Interface Design and Implementation

12

What Can You Do With This Page?



Spring 2012

6.813/6.831 User Interface Design and Implementation

Suggested by Dina Betser

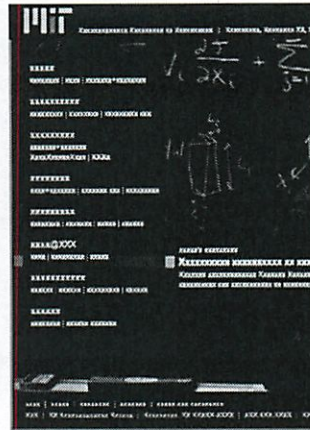
13

If the user is trying to view all the events at once on the CPW website, the user may end up clicking through all the days individually. It turns out that the graphic in the center page is actually a link to a nifty search interface that lets the user look at all the event listings in addition to other cool functionalities, but the graphic doesn't have strong affordances for interaction. It's mostly a big logo, so what does a typical user do? Glance at it and then ignore it, scanning the page instead for things that look like actions, such as the clearly marked hyperlinks at the bottom. The "click here to search" text in the logo **doesn't work**.

It is very easy to miss the search interface altogether because of the poor visibility of the search feature and the lack of affordances that the graphic is a link. (example and explanation due to Dina Betser)

Try It: Playing with Affordances

- Use Javascript to obscure all the text on a page
- Visit several pages, e.g.:
 - MIT's home page
 - 6.813/6.831 home page
- What do the affordances tell you nonverbally?
- Are any of the affordances lying to you?



Spring 2012

6.813/6.831 User Interface Design and Implementation

14

Here's the Javascript code:

```
var result = document.evaluate("//text()", document.body, null,
XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE, null);for (var i = 0; i < result.snapshotLength; ++i) {var
node = result.snapshotItem(i);if ((node.textContent+").match(/\\w/)&&node.parentNode.nodeName !=
\"STYLE\") {node.textContent = node.textContent.replace(/[A-Z0-9]/g, \"X\").replace(/[a-z]/g, \"x\");}}void
0
```

One way to use it is to open your browser's Javascript console and just paste the code in; it will change the current page. Another way to use it is to create a new bookmark in your browser, and use as the URL javascript: followed by the code given above. Clicking on this bookmark will run the Javascript on the current page. (This is called a *bookmarklet*, and it's an one way to modify web pages you don't own.)

FEEDBACK

Spring 2012

6.813/6.831 User Interface Design and Implementation

15

Actions Should Have Immediately Visible Effects

- Low-level feedback

- e.g. push button



- High-level feedback

- model state changes
- new web page starts loading

Spring 2012

6.813/6.831 User Interface Design and Implementation

16

Hand-in-hand with affordances is **feedback**: how the system changes visibly when you perform an action.

When the user invokes a part of the interface, it should appear to respond. Push buttons should depress and release. Scrollbar thumbs and dragged objects should move with the mouse cursor. Pressing a key should make a character appear in a textbox.

Low-level feedback is provided by a view object itself, like push-button feedback. This kind of feedback shows that the interface at least took notice of the user's input, and is responding to it. (It also distinguishes between disabled widgets, which don't respond at all.)

High-level feedback is the actual result of the user's action, like changing the state of the model.

Perceptual Fusion

- Two stimuli within the same perceptual cycle ($T_p \sim 100\text{ms}$ [50-200 ms]) appear **fused**
- Consequences
 - $1/T_p$ frames/sec is enough to perceive a moving picture (10 fps OK, 20 fps smooth)
 - Computer response $< T_p$ feels instantaneous
 - Causality is strongly influenced by fusion

Spring 2012

6.813/6.831 User Interface Design and Implementation

17

One interesting effect of human perceptual system is **perceptual fusion**. Here's an intuition for how fusion works. Our "perceptual processor" runs at a certain frame rate, grabbing one frame (or picture) every cycle, where each cycle takes T_p seconds. Two events occurring less than the cycle time apart are likely to appear in the same frame. If the events are similar – e.g., Mickey Mouse appearing in one position, and then a short time later in another position – then the events tend to *fuse* into a single perceived event – a single Mickey Mouse, in motion.

The cycle time of the perceptual processor can be derived from a variety of psychological experiments over decades of research (summarized in Card, Moran, Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, 1983). 100 milliseconds is a typical value which is useful for a rule of thumb. But it can range from 50 ms to 200 ms, depending on the individual (some people are faster than others) and on the stimulus (for example, brighter stimuli are easier to perceive, so the processor runs faster).

Perceptual fusion is responsible for the way we perceive a sequence of movie frames as a moving picture, so the parameters of the perceptual processor give us a lower bound on the frame rate for believable animation.

10 frames per second is good enough for a typical case, but 20 frames per second is better for most users and most conditions.

Perceptual fusion also gives an upper bound on good computer response time. If a computer responds to a user's action within T_p time, its response feels instantaneous with the action itself. Systems with that kind of response time tend to feel like extensions of the user's body. If you used a text editor that took longer than T_p response time to display each keystroke, you would notice.

Fusion also strongly affects our perception of causality. If one event is closely followed by another – e.g., pressing a key and seeing a change in the screen – and the interval separating the events is less than T_p , then we are more inclined to believe that the first event caused the second.

Response Time

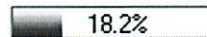
< 0.1 s: seems instantaneous

0.1-1 s: user notices the delay

1-5 s: display busy indicator



> 1-5 s: display progress bar



Spring 2012

6.813/6.831 User Interface Design and Implementation

18

Perceptual fusion provides us with some rules of thumb for responsive feedback.

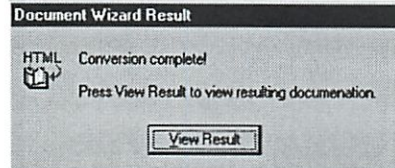
If the system can perform a command in less than 100 milliseconds, then it will seem instantaneous, or near enough. As long as the result of the command itself is clearly visible – e.g., in the user's locus of attention – then no additional feedback is required.

If it takes longer than the perceptual fusion interval, then the user will notice the delay – it won't seem instantaneous anymore. *Something* should change, visibly, within 100 ms, or perceptual fusion will be disrupted. Normally, however, ordinary low-level feedback is enough to satisfy this requirement, such as a push-button popping back, or a menu disappearing.

One second is a typical turn-taking delay in human conversation – the maximum comfortable pause before you feel the need to fill the gap with something, even if it's just "uh" or "um". If the system's response will take longer than a second, then it should display additional feedback. For short delays, the hourglass cursor (or spinning cursor, or throbber icon shown here) is a common design pattern. For longer delays, show a progress bar, and give the user the ability to cancel the command.

Note that progress bars don't necessarily have to be *accurate*. (This one is actually preposterous – who cares about 3 significant figures of progress?) An effective progress bar has to show that progress is being made, and allow the user to estimate completion time at least within an order of magnitude – a minute? 10 minutes? an hour? a day?

Useless Feedback vs. Useful Feedback



Source: Interface Hall of Shame

Spring 2012

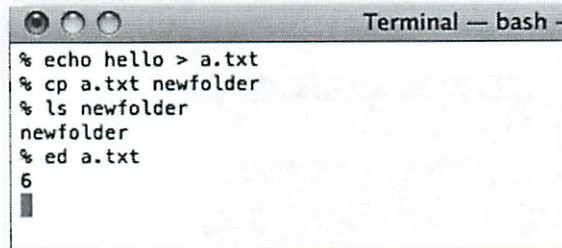
6.813/6.831 User Interface Design and Implementation

19

Feedback is important, but don't overdo it. This dialog box demands a click from the user. Why? Does the interface need a pat on the back for finishing the conversion? It would be better to just skip on and show the resulting documentation.

Example: Unix shell

- Open a command prompt and run:



```
Terminal -- bash -
% echo hello > a.txt
% cp a.txt newfolder
% ls newfolder
newfolder
% ed a.txt
6
█
```

- Think about:
 - affordances
 - feedback

Spring 2012

6.813/6.831 User Interface Design and Implementation

20

Let's summarize with a case study of weak learnability: the Unix command line. Unix may be beautiful for many reasons, but learnability is not one of them.

The **actions** available to the user are completely invisible; the user must recall a command name from memory, along with the syntax for its arguments.

The **state** of the underlying system is likewise mostly hidden. Many users customize their prompts to make some state visible, such as the current directory or the hostname. The contents of the current directory are *not* visible, even though many commands operate on files.

The **feedback** from a command is minimal – in fact, one Unix design principle is that commands should say *nothing* when they succeed. But that's not a good thing. It's true that a generic feedback message like "command completed successfully" would indeed be useless; the subsequent appearance of a command prompt is sufficient feedback that the command has completed. So what kind of visible feedback *would* be useful?

INFORMATION SCENT

Spring 2012

6.813/6.831 User Interface Design and Implementation

21

Information Scent

- Information foraging theory
 - Humans gathering information can be modeled like animals gathering food
 - Constantly evaluating and making decisions to maximize information collected against cost of obtaining it
- Information scent
 - Cues on a link that indicate how profitable it will be to follow the link to its destination

Spring 2012

6.813/6.831 User Interface Design and Implementation

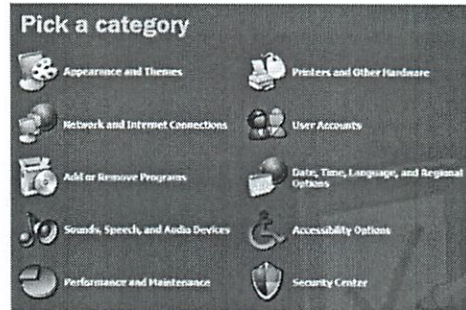
22

Users depend on visible cues to figure out how to achieve their goals with the least effort. For information gathering tasks, like searching for information on the web, it turns out that this behavior can be modeled much like animals foraging for food. An animal feeding in a natural environment asks questions like: Where should I feed? What should I try to eat (the big rabbit that's hard to catch, or the little rabbit that's less filling)? Has this location been exhausted of food that's easy to obtain, and should I try to move on to a more profitable location? **Information foraging theory** claims that we ask similar questions when we're collecting information: Where should I search? Which articles or paragraphs are worth reading? Have I exhausted this source, should I move on to the next search result or a different search? (Pirolli & Card, "Information Foraging in Information Access Environments," *CHI '95*.)

An important part of information foraging is the decision about whether a hyperlink is worth following – i.e., does this smell good enough to eat? Users make this decision with relatively little information – sometimes only the words in the hyperlink itself, sometimes with some context around it (e.g., a Google search result also includes a snippet of text from the page, the site's domain name, the length of the page, etc.) These cues are **information scent** – the visible properties of a link that indicate how profitable it will be to follow the link. (Chi et al, "Using Information Scent to Model User Information Needs and Actions on the Web", *CHI 2001*.)

Give Good Information Scent

- A link should smell like the content it leads to



Spring 2012

6.813/6.831 User Interface Design and Implementation

23

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.

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.

Learnability
 Efficiency
 Errors
 Simplicity

Good & Bad Information Scent

- To learn more about this site, [click here](#)
- Learn more about this site [here](#)
- Learn [more](#) about this site
- Link to this site's [about page](#).

- [Learn more about this site](#)
- [About](#)

Audio and TV
 Books
 Computing
 Fashion
 Furniture
 Gardening

Audio and TV: Camcorders, DVD and Video, Hi-Fi...
 Books: Bestsellers, Factual, Education...
 Computing: Computers, Games, Printers
 Fashion: Mens, Womens, Kids...
 Furniture: Bathrooms, Bedrooms, Kitchen...
 Gardening: Seeds, Plants, Pots

Spring 2012
6.813/6.831 User Interface Design and Implementation
24

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.

Lots of scent but hard to scan

RENT MONKEY

[Home](#) | [Search Listings](#) | [Manage Listings and Profiles](#) | [Residence History](#) | [Browse Residences](#)

Search listings, by MIT students, for MIT students

Many of these listings are unofficial, but they can help guide you towards places with upcoming vacancies. Other listings may be posted by the MIT Off-campus Housing Office.

Advertise an off-campus vacancy or sublet

Want to announce a vacancy or a summer sublet? Manage the listings and residence profiles you've edited on this site. For on-campus lottery and sublets, please visit the [graduate housing website](#).

See what others have said about a residence

Check if other MIT students have written about a particular residence. Look at the rent history of a residence to see how much you should be paying.

Browse where other students are living

Look at where other MIT students are living to guide where you may want to live.

© 2008 GSC HCA, MIT Housing, Robert Wang

[Website Feedback](#)

[Disclaimer](#)

[MIT Rental Guide](#)

[GeoSapla Search](#)

[Ask MIT Housing](#)

[About](#)

More resources:
[GSC HCA](#) | [MIT Housing](#)

Spring 2012

6.813/6.831 User Interface Design and Implementation

25

<http://rentmonkey.mit.edu/account/home>

Hierarchy of Exploration Costs

- Glance
 - affordances, icons, short salient words
- Read
 - description, keywords
- Hover or press
 - cursor change, highlight, tooltip, submenu, preview
- Click through
 - target page, dialog box, or mode
- Invoke
 - feedback effect on the model state

Summary

- Affordances
- Feedback
- Information scent

6.813/6.831 • USER INTERFACE DESIGN AND IMPLEMENTATION

Spring 2012 Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science

HW1: UI HALL OF FAME AND SHAME

Due at 11:59 pm on Sunday, February 19, 2012, by uploading to Stellar.

Help populate the UI Hall of Fame and Shame! Find two examples of user interfaces, one that you consider a good design and one that you consider a bad design. Note that the good design does not have to be uniformly good, since you may discover problems with it on closer inspection. Likewise, the bad design does not have to be uniformly bad. Probably the most interesting examples will be mixed.

Your interfaces might be desktop software, web applications, smartphone apps, consumer devices, car dashboards, building entrances, traffic intersections, shower controls, etc.

For each interface, you should:

- describe the purpose of the interface and its intended users
- analyze its good and bad points of usability with reference to all the dimensions of usability discussed in lecture:
 - learnability
 - efficiency
 - safety
 - you may discuss other aspects of usability if you have space and consider them important
- illustrate your analysis with appropriate screenshots or photographs

Limit to one page of text (roughly 50 lines) for each interface, for a total of 2 pages (100 lines) for your entire report. You can include as many images as are helpful; they don't count toward the page limit.

Grading

Your report will be judged on the following criteria.

- **Completeness.** Don't omit a dimension of usability, and don't overlook an obvious usability issue that even the reader notices.
- **Depth.** "Efficiency is good, because it feels fast to use" is not deep analysis. "I've never made any errors with it" is not deep analysis.
- **Clarity.** The reader should not struggle to understand what you're talking about.
- **Conciseness.** This isn't a HASS class. Unnecessary verbosity will be judged severely.
- **Usability of presentation.** Your report is itself a user interface whose purpose is to convey ideas to a reader. If your report isn't learnable, visible, efficient, and safe, then it will be harder for the reader to use, and it will not demonstrate an ability to apply the ideas of this class.

What to Hand In

Use Stellar to hand in your report as a single PDF file. If your word processor can't generate PDF, there is free software for printing documents out to PDF.

Would it be good to do an entire OS?
-ambitious
Anything else iTunes?
Command prompt?

Camera

↳ - Physical device

Kindle

What am I always saying is poorly designed?

Lenovo's site?

What do I always screw up?

Android!

I can't think of anything else

Focus on the back/switch interface things

↳ use personal pronouns?

This is not super clearly what they asked for...

But should be good

- going above + beyond!

But formal research paper writing style...

Nice focused paper...

HW1: UI Hall of Fame and Shame

Michael Plasmeier

Android and WebOS differ in the way they show running applications and allow users to switch between them. HP's WebOS system presents running applications explicitly as "cards." Google Android 2.3 does not present the concept of running applications explicitly to the user. I will argue that the WebOS model makes it easier for users to control running applications and switch between them, while Android lack of control over running applications confuses users. Therefore I believe that WebOS should be added to the 6.813 UI Hall of Fame; while Android should be added to the Hall of Shame. Both are smartphone operating systems seeking a wide range of users.

HP WebOS

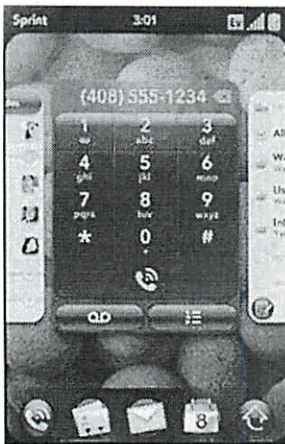


Figure 1 Card View

HP WebOS uses a card based metaphor to display running applications. Each running application is represented by a card in the "card view." The single, main button on the device brings up "card view." In "card view," the previously running application is minimized into a card that takes up ~80% of the screen. From here a user can swipe left and right to see the other applications that are currently running. If a user wishes to go back to that running app, the user can either tap the card or flick it towards the bottom of the screen. A user can close an app by flicking the card up and off the top of the screen. An application can have multiple cards, just like a Windows application can have multiple Windows. For example, the email application launches a new card when the user taps the "new message" button. This allows the user to move between the compose card and the messages card to review a previously sent email while composing a new email.

This effect is highly learnable. When the user firsts uses a WebOS phone, a short video featuring a ball of light teaches users how to use their phone.¹ In addition,



Figure 2 Too Many Cards

the button to bring up card view is the main button on the device. This leads users to try to click on the button to make something happen. This button is in the same place as the iPhone, which creates consistency between phones. In card view, a little bit of the left and right of each card is visible, letting users know they can scroll left and/or right to see the other cards. An error message appears when a user has too many cards open, letting the user know they can swipe a card upward to close applications. No applications are automatically closed.

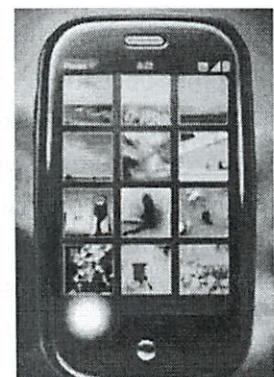


Figure 3 Intro Video on Palm Pre

¹ <http://www.youtube.com/watch?v=KCjhqbVM-XY>

The system is very efficient. It is very easy to switch between applications. For example, while a user is writing an email, they can hit the center button, swipe left, and tap on their calendar to view it. They can then do the reverse and be brought back to their email in progress. The compose card will stay open, unless a user manually closes it. Each browser page is a multiple card – there is no tabbed browsing. The downside is that a user will get an error message when they have too many cards open and must manually close cards.

The system is safe. Cards remain open, unless a user chooses to close them. This allows a user to keep something open to refer to later.

Google Android 2.3



Figure 4 Android's Recent Items Menu

Google Android 2.3, meanwhile, does not make the list of running applications explicit to the user. In the stock version of Android 2.3 applications may be closed at any time. Android primarily expects users to reopen running application through the normal methods – as though they were starting the application for the first time. Android does have a recent application menu that shows the most recent 8 applications. This menu is accessible by pressing and holding the physical home key. Applications are treated as a whole; only one entry is shown. For example, the compose feature of an email application is not shown differently, and only one entry for a browser is shown, requiring browser vendors to implement tabs inside the application.

The most inconsistent and confusing UI control on Android 2.3 devices is the hardware back button. This button alternatively switches between going back in the application, going to a different application, or closing the application. For example, if a user clicks on a link in the Twitter application, the browser opens. When the user clicks the back button, the browser closes and the user is taken back to the Twitter page. This behavior feels natural. However, when the user is in the email application, and opens a message; clicking on the back key returns the user to the message list. This also feels natural. However, these two features are inconsistent. This sometimes leads to problems. For example, if a user leaves the email app open while looking at a message, open another app, then reopens the email application, the email app goes straight to showing the email message that was last opened. At that point the back key would close the email application, whereas the user might have wanted to return to the message list; which was the previous behavior in the back button.

The learnability of Android is poor. The long press for home is not visible and is not described during initial setup. Whereas, the back button on a device is highly visible, its behavior is inconsistent and thus difficult to learn. It is not clear that certain apps are running or not or even that an app might remain running when another app is opened. Android wants users not to worry about this, however this creates side effects. The proliferation of Task Killers for Android shows that many users do not trust or rely on this feature.

The process is not efficient. Take for example, the task of a user wanting to check their calendar while writing an email. Because the user likely does not

INSTALLS:
10,000,000 - 50,000,000

Figure 5 Advanced Task Killer has > 10 million installs

know about the recent apps page, the user must close the email app in order to get to the home screen's launcher. This causes the email to be saved in the drafts folder. The user then launches the calendar app from the launcher. When the user is done using the calendar, they use the back button to exit to the menu. They must then relaunch the email app, which may open directly with the saved draft. These are a lot of steps and the user is tapping in many different areas on the screen.

This approach is not inherently safe. It requires the app to proactively save its state in case that it is terminated. There is no notion of keeping apps open for a user to refer to later.

Based simply on how these two mobile OSes manage applications, HP WebOS should be placed in the Hall of Fame while Android 2.3, should be remanded to the UI Hall of Shame. There are also more things that HP WebOS does well, such as notifications and contact management that can be explored in future papers.

User Centered Design

GA 1 is at

Hall of Fame/shame Tabbed Browsing

- Remove clutter from taskbar
- Can arrange into task groups
- Can close whole window at once
- but they don't scale
 - Stacking tabs screws up the metaphor
 - Changes the order of the other tabs
 - moves what you just clicked on away from mouse
 - ↳ generally a very bad idea
- Can overflow tabs into a list
 - ↳ Eclipse
 - the non hold face ones are visible
 - ↳ confusing
 - Can also type a search and it filters list
 - ↳ CTRL + E if you know it
 - Very hard to find
 - Very few affordances

②

- Could use a magnifying glass
 - CTRL + E is same in FF ~~etc~~
 - but in FF it searches Google
 - This only does prefix search
-

Nanoquiz

Feedback

key clicks

dragged objects

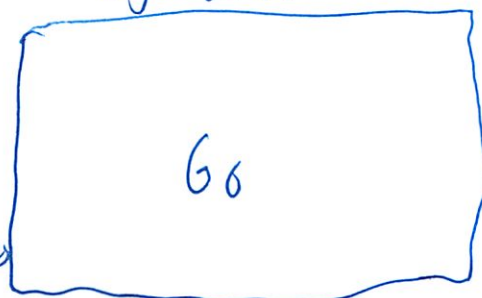
Perceptual fusion

- related to speed feedback appears
- related to frame rate of movies
(iso speed we see something)

Scrollbar

- arrow buttons
- thumb along track
- Thumb textured for dragging
- > You can create a big scrollbar
 - people won't recognize it
- Some w/ buttons

too big!
lose affordances



③

Touchscreen

- no scroll bar
 - but see half line
 - lets you know you can ~~scroll~~ scroll
 - Or it appears for a few seconds
-

Today Iterative design

User + task analysis

- why is this UI so good?
- break it down
- assign a name + reason
- then can stick stuff back together
- what engineering edu is all about

Class Project

User-Centered Design

- Iterative design
- Early focus on users + tasks
- Constant evaluation

④

Iterative Design

Not waterfall model

Requirements

↖
a bit of
feedback

Design

↖
Code

↖
Integration

↖
Acceptance

↖
~~Release~~
Release

Users only at start + end

UI design is risky

- don't know if it will work

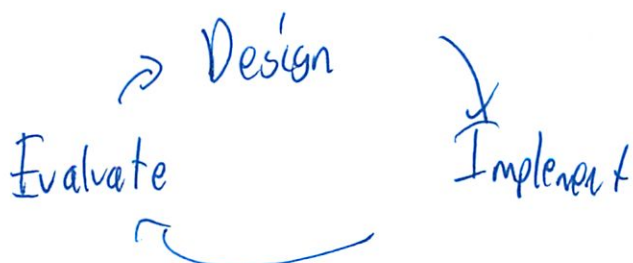
Get Users involved early on

UI flaws change design a lot

- messy

- lot of work

Iterative



⑤

Can build low end / low fidelity prototypes

- paper mockup
- sketches
- computer testing

keep early phases cheap - so can try a lot of them
So only mature iterations are implemented + released

- Or people won't come back
- Start ups should only release good code

It's also cheaper now to build UI on computer

- Interface builder
- But still hard to do the backend changes

People give more feedback w/ paper prototype

- don't want you to do extra work
- So they give you better feedback
- architects know this

You have to throw away the paper

- if you do a fast UI prototype \rightarrow you'll keep it
L is prob crappy - since done fast Lost of time

6

IBM's Olympic Message System

lots of iterations

Simulations - Wizard of Oz

- Someone reads from a script
- human simulating the back end

~~But forgot~~ about not - English speakers
Thought

- upper + lower case

Know Your User

Prob not like you

- Age, Gender, Culture, Language
- Edu
- Physical Limitations
- Computer experience
- Motivation, attitude
- Domain + App experience
- work env + social contexts
- Relationships w/ other people or systems

⑦

Don't set user requirements

↳ Be realistic to what they actually are

Can be diff type of roles

People can be diff characteristics

Exercise Drug Design

~~Who is the drug~~

Who are the users?

(This seems more like if you are going into a client meeting - what do you ask?)

- Purpose of system

- Users

 - what experience?

 - comfortableness w/ ~~system~~ computers

 - Clerks or doctors

- Public Facing

④

Who are the users?

Sysadmins

Pharmacists

Doctors

Canteen fitters?

~~Users~~ Consumers / patients

Regulators

Med students

Manufacturers

Researchers

Insurance Cos

- Caregivers

- Age

- Literacy levels

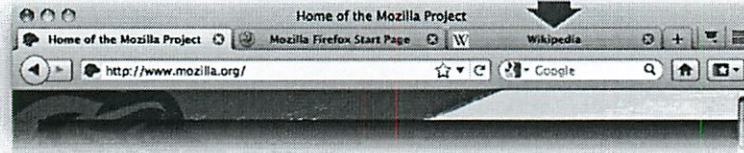
- Lang

Who am I building ~~the~~ this for? Why?

L5: User-Centered Design

- GR1 (project proposal) out, due next Sun

UI Hall of Fame or Shame?



Spring 2011

6.813/6.831 User Interface Design and Implementation

2

Today's candidate for the User Interface Hall of Fame is **tabbed browsing**, a feature found in almost all web browsers. With tabbed browsing, multiple browser windows are grouped into a single top-level window and accessed by a row of tabs. You can open a hyperlink in a new tab by choosing that option from the right-click menu.

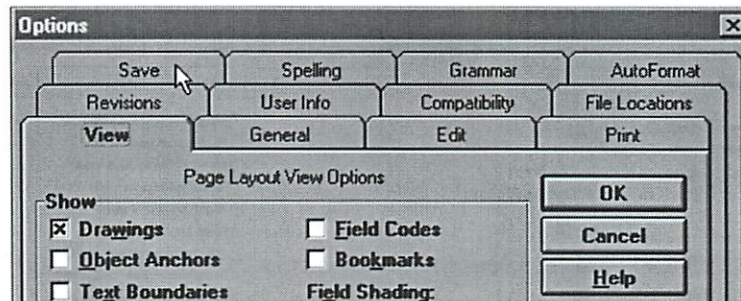
Tabbed browsing neatly solves a scaling problem in the Windows taskbar. If you accumulate several top-level windows, they cease to be separately clickable buttons in the taskbar and merge together into a single button with a popup menu. So your browser windows become **less visible** and **less efficient** to reach.

Tabbed browsing solves that by creating effectively a separate task bar specialized to the web browser. But it's even better than that: you can open multiple top-level browser windows, each with its own set of tabs. Each browser window can then be dedicated to a particular task, e.g. apartment hunting, airfare searching, programming documentation, web surfing. It's an easy and natural way for you to create task-specific groupings of your browser windows. That's what the Windows task bar tries to do when it groups windows from the same application together into a single popup menu, but that simplistic approach doesn't work at all because the Web is such a general-purpose platform.

Another neat feature of tabbed browsing is that you can bookmark a set of tabs so you can recover them again later – a nice **shortcut** for task-oriented users.

What are the downsides of tabbed browsing? For one thing, you can't compare the contents of one tab with another. External windows let you do this by resizing and repositioning the windows.

Hall of Shame



Spring 2011

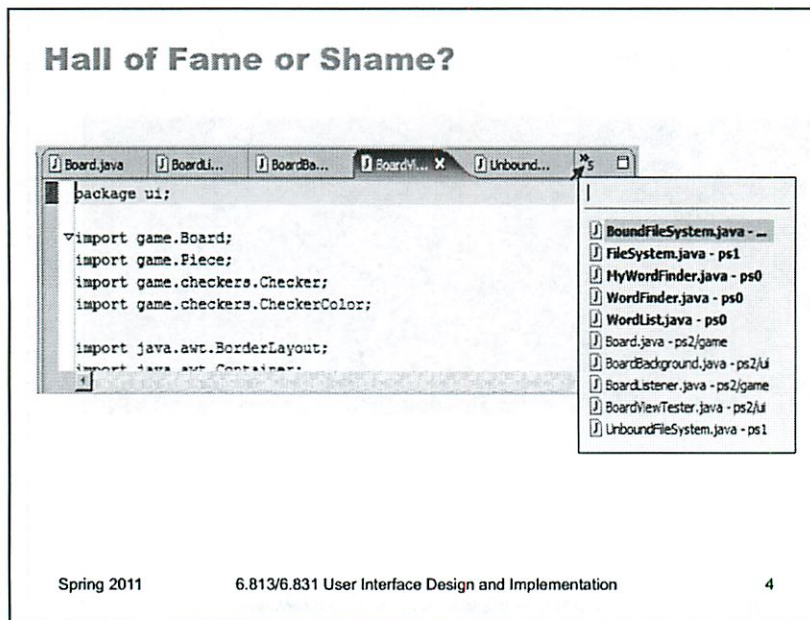
6.813/6.831 User Interface Design and Implementation

3

Another problem is that tabs don't really scale up either – you can't have more than 5-10 without shrinking their labels so much that they're unreadable. Some designers have tried using **multiple rows of tabs**, but if you stick slavishly to the tabbing metaphor, this turns out to be a horrible idea. Here's the Microsoft Word 6 option dialog. Clicking on a tab in a back row (like Spelling) has to move the whole row forward in order to maintain the tabbing metaphor. This is disorienting for two reasons: first, because the tab you clicked on has leaped out from under the mouse; and second, because other tabs you might have visited before are now in totally different places. Some plausible solutions to these problems were proposed in class – e.g., color-coding each row of tabs, or moving the front rows of tabs *below* the page. Animation might help too. All these ideas might reduce disorientation, but they involve tradeoffs like added visual complexity, greater demands on screen real estate, or having to move the page contents in addition to the tabs. And none of them prevent the tabs from jumping around, which is a basic problem with the approach.

As a rule of thumb, only one row of tabs really works if you want the tabs to tie directly to the panel, and the number of tabs you can fit in one row is constrained by the screen width and the tab label width. Most tabbing controls can scroll the tabs left to right, but scrolling tabs is definitely slower than picking from a popup menu.

In fact, the Windows task bar actually scales better than tabbing does, because it doesn't have to struggle to maintain a metaphor. The Windows task bar is just a row of buttons. Expanding the task bar to show two rows of buttons puts no strain on its usability, since the buttons don't have to jump around.



Here's how Eclipse tries to address the tab scaling problem: it shows a few tabs, and the rest are found in a pulldown menu on the right end of the tab bar.

This menu has a couple of interesting features. First, it offers **incremental search**: typing into the first line of the menu will narrow the menu to tabs with matching titles. If you have a very large number of tabs, this could be a great shortcut. But it doesn't communicate its presence very well. I used Eclipse for months, and didn't even notice this feature until I started carefully exploring the tab interface for this Hall of Fame & Shame discussion.

Second, the menu tries to distinguish between the visible tabs and the hidden tabs using boldface. Quick, before studying the names of the tabs carefully -- which do you think is which? Was that a good decision?

Picking an item from the menu will make it appear as a tab -- replacing one of the tabs that's currently showing. Which tab will get replaced? It's not immediately clear.

The key problem with this pulldown menu is that it completely disregards the **natural, spatial mapping** that tabs provide. The menu's order is unrelated to the order of the visible tabs; instead, the menu is alphabetical, but the tabs themselves are listed in order of recent use. If you choose a hidden tab, it replaces the least recently used visible tab. LRU is a great policy for caches. Is it appropriate for frequently-accessed menus? Probably not, because it interferes with users' spatial memory.

Today's Topics

- Iterative design
- User & task analysis
- Class project

User-Centered Design

- Iterative design
- Early focus on users and tasks
- Constant evaluation

Spring 2011

6.813/6.831 User Interface Design and Implementation

8

The standard approach to designing user interfaces is **user-centered design**, which has three components. We'll talk about the first two today; we'll defer evaluation (testing with users) until a later lecture.

ITERATIVE DESIGN

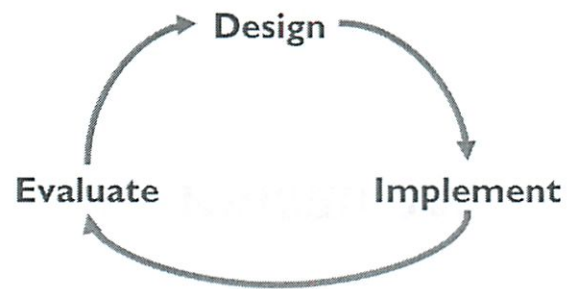
Spring 2011

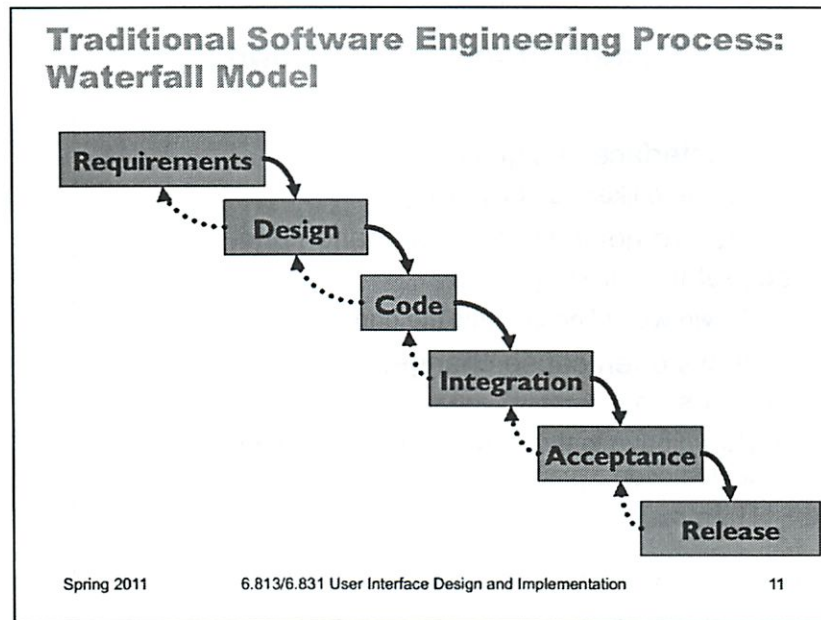
6.813/6.831 User Interface Design and Implementation

9

Iterative Design

- Rinse, lather, repeat!





Let's contrast the iterative design process against another way. The **waterfall model** was one of the earliest carefully-articulated design processes for software development. It models the design process as a sequence of stages. Each stage results in a concrete product – a requirements document, a design, a set of coded modules – that feeds into the next stage. Each stage also includes its own **validation**: the design is validated against the requirements, the code is validated (unit-tested) against the design, etc.

The biggest improvement of the waterfall model over previous (chaotic) approaches to software development is the discipline it puts on developers to **think first, and code second**. Requirements and designs generally precede the first line of code.

If you've taken a software engineering course, you've experienced this process yourself. The course staff probably handed you a set of requirements for the software you had to build --- e.g. the specification of a chat client or AntiBattleship. (In the real world, identifying these requirements would be part of your job as software developers.) You were then expected to meet certain milestones for each stage of your project, and each milestone had a concrete product: (1) a design document; (2) code modules that implemented certain functionality; (3) an integrated system.

Validation is not always sufficient; sometimes problems are missed until the next stage. Trying to code the design may reveal flaws in the design – e.g., that it can't be implemented in a way that meets the performance requirements. Trying to integrate may reveal bugs in the code that weren't exposed by unit tests. So the waterfall model implicitly needs **feedback between stages**.

The danger arises when a mistake in an early stage – such as a missing requirement – isn't discovered until a very late stage – like acceptance testing. Mistakes like this can force costly rework of the intervening stages. (That box labeled "Code" may look small, but you know from experience that it isn't!)

Waterfall Model Is Bad for UI Design

- User interface design is risky
 - So we're likely to get it wrong
- Users are not involved in validation until acceptance testing
 - So we won't find out until the end
- UI flaws often cause changes in requirements and design
 - So we have to throw away carefully-written and tested code

Spring 2011

6.813/6.831 User Interface Design and Implementation

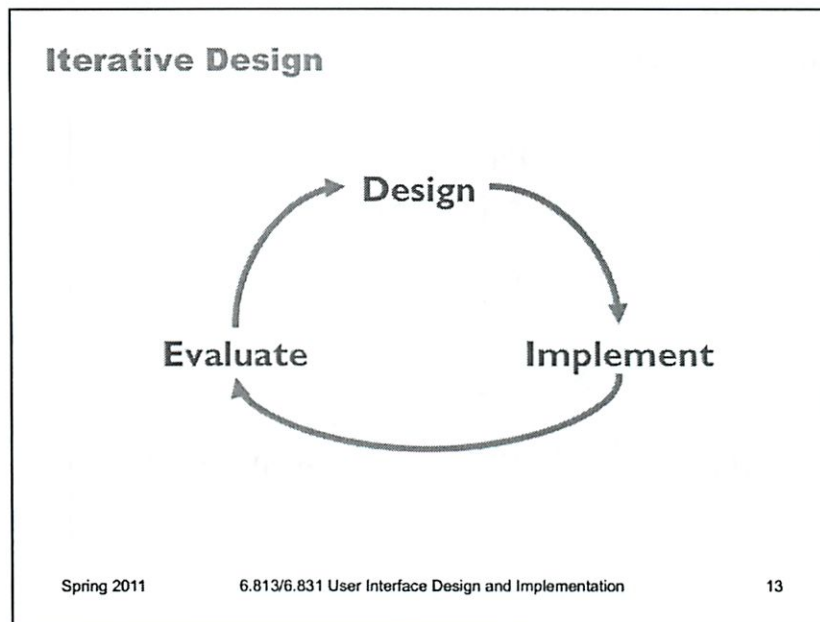
12

Although the waterfall model is useful for some kinds of software development, it's very poorly suited to user interface development.

First, UI development is inherently **risky**. UI design is hard for all the reasons we discussed in the previous lecture. (You are not the user; the user is always right, except when the user isn't; users aren't designers either.) We don't (yet) have an easy way to predict how whether a UI design will succeed.

Second, in the usual way that the waterfall model is applied, **users appear in the process in only two places**: requirements analysis and acceptance testing. Hopefully we asked the users what they needed at the beginning (requirements analysis), but then we code happily away and don't check back with the users until we're ready to present them with a finished system. So if we screwed up the design, the waterfall process won't tell us until the end.

Third, when UI problems arise, they often **require dramatic fixes**: new requirements or new design. We saw in Lecture 1 that slapping on patches doesn't fix serious usability problems.



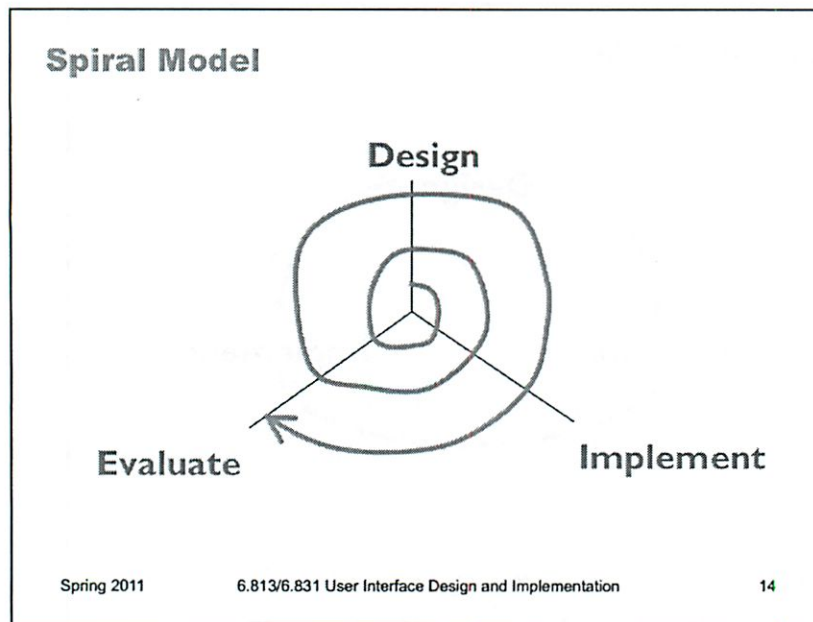
Iterative design offers a way to manage the inherent risk in user interface design. In iterative design, the software is refined by repeated trips around a design cycle: first imagining it (design), then realizing it physically (implementation), then testing it (evaluation).

In other words, we have to admit to ourselves that we aren't going to get it right on the first try, and plan for it. Using the results of evaluation, we redesign the interface, build new prototypes, and do more evaluation. Eventually, hopefully, the process produces a sufficiently usable interface.

Sometimes you just iterate until you're satisfied or run out of time and resources, but a more principled approach is to set usability goals for your system. For example, an e-commerce web site might set a goal that users should be able to complete a purchase in less than 30 seconds.

Many of the techniques we'll learn in this course are optimizations for the iterative design process: design guidelines reduce the number of iterations by helping us make better designs; cheap prototypes and discount evaluation techniques reduce the cost of each iteration. But even more important than these techniques is the basic realization that in general, **you won't get it right the first time**. If you learn nothing else about user interfaces from this class, I hope you learn this.

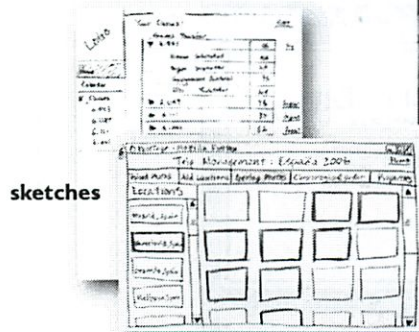
You might object to this, though. At a high level, iterative design just looks like the worst-case waterfall model, where we made it all the way from design to acceptance testing before discovering a design flaw that **forced** us to repeat the process. Is iterative design just saying that we're going to have to repeat the waterfall over and over and over? What's the trick here?



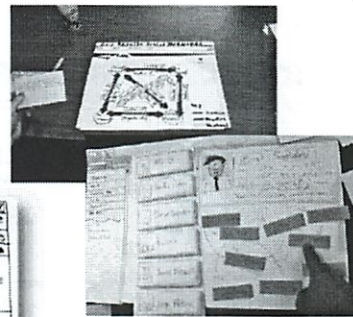
The **spiral model** offers a way out of the dilemma. We build room for several iterations into our design process, and we do it by making the early iterations as cheap as possible.

The radial dimension of the spiral model corresponds to the **cost** of the iteration step – or, equivalently, its **fidelity** or **accuracy**. For example, an early implementation might be a paper sketch or mockup. It's low-fidelity, only a pale shadow of what it would look and behave like as interactive software. But it's incredibly cheap to make, and we can evaluate it by showing it to users and asking them questions about it.

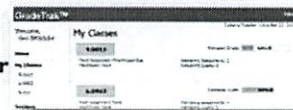
Early Prototyping



sketches



paper prototypes



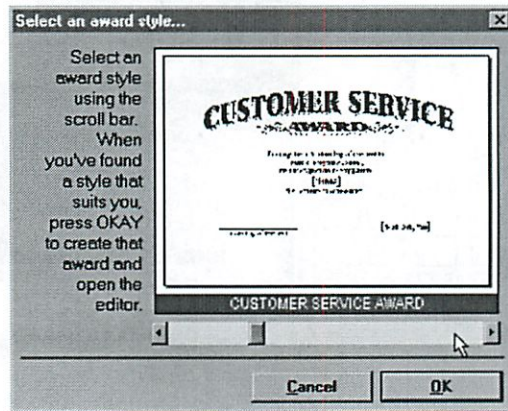
computer
mockups

Spring 2012

6.813/6.831 User Interface Design and Implementation

15

Early Prototypes Can Detect Usability Problems



Spring 2011

6.813/6.831 User Interface Design and Implementation

16

Remember this Hall of Shame candidate from the first lecture? This dialog's design problems would have been easy to catch if it were only tested as a simple paper sketch, in an early iteration of a spiral design. At that point, changing the design would have cost only another sketch, instead of a day of coding.

Iterative Design of User Interfaces

- Early iterations use cheap prototypes
 - **Parallel design** is feasible: build & test multiple prototypes to explore design alternatives
- Later iterations use richer implementations, after UI risk has been mitigated
- More iterations generally means better UI
- Only mature iterations are seen by the world

Spring 2011

6.813/6.831 User Interface Design and Implementation

17

Why is the spiral model a good idea? Risk is greatest in the early iterations, when we know the least. So we put our least commitment into the early implementations. Early prototypes are made to be thrown away. If we find ourselves with several design alternatives, we can build multiple prototypes (**parallel design**) and evaluate them, without much expense.

After we have evaluated and redesigned several times, we have (hopefully) learned enough to avoid making a major UI design error. Then we actually implement the UI – which is to say, we build a prototype that we intend to keep. Then we evaluate it again, and refine it further.

The more iterations we can make, the more refinements in the design are possible. We're hill-climbing here, not exploring the design space randomly. We keep the parts of the design that work, and redesign the parts that don't. So we should get a better design if we can do more iterations.

Case Study of User-Centered Design: The Olympic Message System

- Cheap prototypes
 - Scenarios
 - User guides
 - Simulation (Wizard of Oz)
 - Prototyping tools (IBM Voice Toolkit)
- Iterative design
 - 200 (!) iterations for user guide
- Evaluation at every step
- You are not the user
 - Non-English speakers had trouble with alphabetic entry on telephone keypad



Spring 2011

6.813/6.831 User Interface Design and Implementation

18

The Olympic Message System is a classic demonstration of the effectiveness of user-centered design (Gould et al, "The 1984 Olympic Message System", CACM, v30 n9, Sept 1987). The OMS designers used a variety of cheap prototypes: scenarios (stories envisioning a user interacting with the system), manuals, and simulation (in which the experimenter read the system's prompts aloud, and the user typed responses into a terminal). All of these prototypes could be (and were) shown to users to solicit reactions and feedback.

Iteration was pursued aggressively. The user guide went through 200 iterations!

The OMS also has some interesting cases reinforcing the point that the designers cannot rely entirely on themselves for evaluating usability. Most prompts requested numeric input ("press 1, 2, or 3"), but some prompts needed alphabetic entry ("enter your three-letter country code"). Non-English speakers – particularly from countries with non-Latin languages – found this confusing, because, as one athlete reported in an early field test, "you have to read the keys differently." The designers didn't remove the alphabetic prompts, but they did change the user guide's examples to use only uppercase letters, just like the telephone keys.

A video about OMS can be found on YouTube (<http://youtube.com/watch?v=W6UYpXc4czM&feature=related>). Check it out – it includes a mime demonstrating the system.

USER & TASK ANALYSIS

Spring 2012

6.813/6.831 User Interface Design and Implementation

19

Know Your User

- Questions to ask
 - Age, gender, culture, language
 - Education (literacy? numeracy?)
 - Physical limitations
 - Computer experience (typing? mouse?)
 - Motivation, attitude
 - Domain experience
 - Application experience
 - Work environment and other social context
 - Relationships and communication patterns with other people
- Pitfall
 - describing what you *want* your users to be, rather than what they *actually are*
 - "Users should be literate in English, fluent in spoken Swahili, right-handed, and color-blind"

Spring 2011

6.813/6.831 User Interface Design and Implementation

20

The reason for user analysis is straightforward: since you're not the user, you need to find out who the user actually is.

User analysis seems so obvious that it's often skipped. But failing to do it explicitly makes it easier to fall into the trap of assuming every user is like you. It's better to do some thinking and collect some information first.

Knowing about the user means not just their individual characteristics, but also their situation. In what environment will they use your software? What else might be distracting their attention? What is the social context? A movie theater, a quiet library, inside a car, on the deck of an aircraft carrier; environment can place widely varying constraints on your user interface.

Other aspects of the user's situation include their relationship to other users in their organization, and typical communication patterns. Can users ask each other for help, or are they isolated? How do students relate differently to lab assistants, teaching assistants, and professors?

Many problems in user and task analysis are caused by jumping too quickly into a requirements mindset. In user analysis, this sometimes results in wishful thinking, rather than looking at reality. Saying "OMS users *should* all have touchtone phones" is stating a **requirement**, not a characteristic of the existing users. One reason we do user analysis is to see whether these requirements are actually satisfied, or whether we'd have to add something to the system to make sure it's satisfied. For example, maybe we'd have to offer touchtone phones to every athlete's friends and family...

Multiple Classes of Users

- Many applications have several kinds of users
 - By role (student, teacher)
 - By characteristics (age, motivation)
- Example: Olympic Message System
 - Athletes
 - Friends & family
 - Telephone operators
 - Sysadmins

Spring 2011

6.813/6.831 User Interface Design and Implementation

21

Many, if not most, applications have to worry about multiple classes of users.

Some user groups are defined by the roles that the user plays in the system: student, teacher, reader, editor.

Other groups are defined by characteristics: age (teenagers, middle-aged, elderly); motivation (early adopters, frequent users, casual users). You have to decide which user groups are important for your problem, and do a user analysis for every class.

The Olympic Message System case study we saw in a previous lecture identified several important user classes by role.

Exercise: Drug Database

- Suppose a company has contracted you to design “a database of drugs”
- With your neighbor:
 - Decide on questions you should ask the company or its users
 - Make up plausible answers to your questions



Identify the User's Tasks and Goals

- Identify the individual tasks involved in the problem
- Decompose them into subtasks
- (think about) abstracting them into goals
- Example: Olympic Message System
 - send message to an athlete
 - find out if I have messages
 - listen to my messages

The best sources of information for task analysis are user interviews and direct observation. Usually, you'll have to observe how users *currently* perform the task. For the OMS example, we would want to observe athletes interacting with each other, and with family and friends, while they're training for or competing in events. We would also want to interview the athletes, in order to understand better their goals in the task.

Questions to Ask About a Task

- Why is the task being done?
- What does the user need to know or have before doing the task?
- Where is the task performed?
 - At a kiosk, standing up
- What is the environment like? Noisy, dirty, dangerous?
 - Outside
- How often is the task performed?
 - Perhaps a couple times a day
- What are its time or resource constraints?
 - A minute or two (might be pressed for time!)
- How is the task learned?
 - By trying it
 - By watching others
 - Classroom training? (probably not)
- What can go wrong? (Exceptions, errors, emergencies)
 - Enter wrong country code
 - Enter wrong user name
 - Get distracted while recording message
- Who else is involved in the task?

Spring 2011

6.813/6.831 User Interface Design and Implementation

25

There are lots of questions you should ask about each task. Here are a few, with examples relevant to the OMS send-message task. Collecting this information about tasks helps inform your design.

Common Errors in Task Analysis

- Thinking from the system's point of view, rather than the user's
 - "Notify user about appointment"
 - vs. "Get a notification about appointment"
- Fixating too early on a UI design vision
 - "The system bell will ring to notify the user about an appointment..."
- Boggling down in *what* users do now (**concrete** tasks), rather than *why* they do it (**essential** tasks)
 - "Save file to disk"
 - vs. "Make sure my work is kept"
- Duplicating a bad existing procedure in software
- Failing to capture good aspects of existing procedure

Spring 2011

6.813/6.831 User Interface Design and Implementation

26

The requirements mindset can also affect task analysis. If you're writing down tasks from the system's point of view, like "Notify user about appointment", then you're writing **requirements** (what the system should do), not **tasks** (what the user's goals are). Sometimes this is merely semantics, and you can just write it the other way; but it may also mean you're focusing too much on what the system *can* do, rather than what the user *wants*. Tradeoffs between user goals and implementation feasibility are inevitable, but you don't want them to dominate your thinking at this early stage of the game.

Task analysis derived from observation may give too much weight to the way things are currently done. A task analysis that breaks down the steps of a current system is **concrete**. For example, if the *Log In* task is broken down into the subtasks *Enter username* and *Enter password*, then this is a concrete task relevant only to a system that uses usernames and passwords for user identification. If we instead generalize the *Log In* task into subtasks *Identify myself* and *Prove my identity*, then we have an **essential** task, which admits much richer design possibilities when it's time to translate this task into a user interface.

A danger of concrete task analysis is that it might preserve tasks that are inefficient or could be done a completely different way in software. Suppose we did a task analysis by observing users interacting with paper manuals. We'd see a lot of page flipping: "Find page N" might be an important subtask. We might naively conclude from this that an online manual should provide really good mechanisms for paging & scrolling, and that we should pour development effort into making those mechanisms as fast as possible. But page flipping is an artifact of physical books! It would pay off much more to have fast and effective searching and hyperlinking in an online manual. That's why it's important to focus on **why** users do what they do (the essential tasks), not just what they do (the concrete tasks).

An incomplete task analysis may fail to capture important aspects of the existing procedure. In one case, a dentist's office converted from manual billing to an automated system. But the office assistants didn't like the new system, because they were accustomed to keeping important notes on the paper forms, like "this patient's insurance takes longer than normal." The automated system provided no way to capture those kinds of annotations. That's why interviewing and observing real users is still important, even though you're observing a concrete task process.

Exercise: Elevator Task Analysis

- Suppose we're designing the Student Center elevator interface
- What are the tasks?

Techniques for Understanding Users & Tasks

- Interviews & observation
- Contextual inquiry technique
 - Interviews & observation conducted “in context”, i.e. with real people dealing with the real problem in the real environment
 - Establish a master-apprentice relationship
 - User shows how and talks about it
 - Interviewer watches and asks questions
- Participatory design technique
 - Including a user directly on the design team

Spring 2011

6.813/6.831 User Interface Design and Implementation

29

The best sources of information for task analysis are user interviews and direct observation. Usually, you'll have to observe how users *currently* perform the task. For the OMS example, we would want to observe athletes interacting with each other, and with family and friends, while they're training for or competing in events. We would also want to interview the athletes, in order to understand better their goals in the task.

Contextual inquiry is a technique that combines interviewing and observation, in the user's actual work environment, discussing actual work products. Contextual inquiry fosters strong collaboration between the designers and the users. (Wixon, Holtzblatt & Knox, “Contextual design: an emergent view of system design”, CHI '90)

Participatory design includes users directly on the design team – participating in the task analysis, proposing design ideas, helping with evaluation. This is particularly vital when the target users have much deeper domain knowledge than the design team. It would be unwise to build an interface for stock trading without an expert in stock trading on the team, for example.

CLASS PROJECT

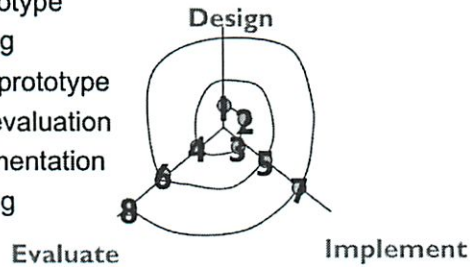
Spring 2012

6.813/6.831 User Interface Design and Implementation

30

User-Centered Design in 6.813/6.831: Group Project

1. GR1: Proposal & user/task analysis
2. GR2: Design sketches
3. GR3: Paper prototype
4. GR3: User testing
5. GR4: Computer prototype
6. HW2: Heuristic evaluation
7. GR5: Full implementation
8. GR6: User testing



Spring 2011

6.813/6.831 User Interface Design and Implementation

31

The term project's milestones are designed to follow a user-centered design process:

1. task and user analysis (1 week): collecting the requirements for the UI, which we'll discuss in the next lecture.
2. design sketches (1 week): paper sketches of various UI designs.
3. paper prototype (1 week): an interactive prototype made of paper and other cheap physical materials.
4. user testing (1 week): during the paper prototype assignment, you'll test your prototype on your classmates.
5. computer prototype (2 weeks): an incomplete but interactive software prototype.
6. heuristic evaluation (1 week): we'll exchange implementation prototypes and evaluate them as usability experts would.
7. full implementation (3 weeks): you'll build a real implementation that you plan to keep.
8. user testing (1 week): you'll test your implementation against users and refine it.

Notice that the part you probably did in your software engineering class is step 7 – which is only one milestone in this class!

Summary

- Iterative design
- User & task analysis

GR1 due Sun

No class Fri

PS1/RS1 out Mon due Sun

↑
u grads ↑
grads

Hall of Fame / Shame Modal Dialog Box

Prevents ya from interacting w/ parents window

Can now look at other apps — not before

So can't see extra context

But don't want to accidentally change context

Look identical to some modalless dialog box

— So we don't know which is which

Mac's Modal Dialog is a sheet that rolls down from the app's title bar

Consistency + affordances

②

Nanoquiz

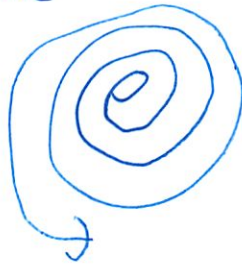
Waterfall model not part user-centered design

- risky design space
- substantial user involvement
- you are not user

① All correct

Spiral started w/ sw engineering
↳ moved to UI

Spiral is outward



Today: Chunks

Pointing + Steering

Shortcuts

Keystroke Level Model

③

Chunking

Originally a unit of memory or perception

Depends both on presentation and what you already know

Can remember chunks better than ~~original~~ just string
esp if resemble rich semantic contexts (brands)

Remembering chess configs

Chess masters far better able for real games

Could chunk into patterns

- castling
- rook rows

Both masters + novices no good at
random, implausible to have

Working memory

7 ± 2 chunks for 10 sec

Unless repeat over + over

4

Long term memory - elaborative rehearsal
- build new connections/chunks

fs

- should display MB, GB
- doesn't display units
- no commas or spaces
 - parse faster
- chunk long strings

Pointing + Steering

Fitts' Law

- ~~all~~ doesn't depend on context
- time it takes you to hit a target
 - w/ finger or mouse
- depends on
 - size of target
 - distance of target $\propto \log$ related

5

Since can move hand fast, but then must fire ~~there~~ there

- 2nd motion bounded by acc error from 1st motion = e_D

$e_D < S$ to hit target

Target at edge of screen \rightarrow since ∞ size since can overshoot

Corners are even better

- Offices $\textcircled{\#}$ menu is in corner

- Just a bar item on top

| |
|------|
| Edit |
|------|

constrained by smallest width

Pie menu

- Ring

- can overshoot



Steering Law

Need to go through "tunnel"

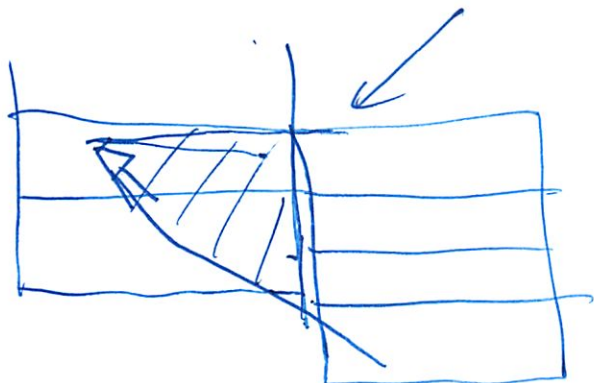
Now linear - not log

Exponentially longer than log

④

This is cascading menus

Instead can create triangular activation error



If don't - efficiency problem

Rules

- Make freq used targets big
- Rt used targets near each other
- Use edges + corners
- Avoid steering tasks

Others

- Fill forms w/ defaults
 - ↳ select everything that is in the field
 - ↳ "pending ~~data~~ delete"
- Offer freq/recent used
 - but keep it in the main corpus

⑦

Auto complete

- see possibilities

Multiple selection

- do many at once

Keystroke Level Models

- Predictive evaluation (abstract idea)
- Save seconds off common tasks
 - Can spend a lot of time
- Have model of human
 - Simulate against this model
- It works well in ~~can~~ other fields
 - like bridge engineering
- So use data like 7 ± 2 chunks model
- Theory provides example of what is causing the problem

②

Simple approach: keystroke Level Model (KLM)

key stroke

Button press/release w/ mouse

Point w/ mouse

D can live w/ mouse

Many hands b/w mouse + keyboard

Mentally prepare

- not visible to analyst

- not learning UI

This is for expert users

Encode a method as seq of those low level operators

Use heuristics to insert thinking

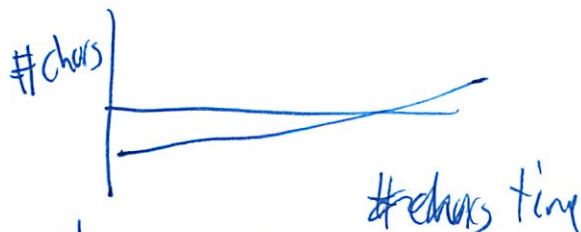
Have estimates from previous experiments

- or rough estimate \rightarrow key stroke = 1 sec

So ~~all~~
to delete

Method 1
5 sec

Method 2
3 sec + n

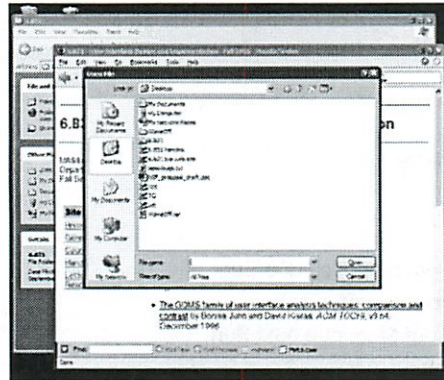


We are always making decisions b/w

L6: Efficiency

- GR1 (project proposal) due Sun
- no class this Friday
- PS1/RS1 out Mon, due next Sun

UI Hall of Fame or Shame?



Suggested by Vikki Chou

Spring 2012

6.813/6.831 User Interface Design and Implementation

2

Today's candidate for the Hall of Fame or Shame is the **modal dialog box**.

A modal dialog box (like the File Open dialog seen here) prevents the user from interacting with the application that popped it up.

Modal dialogs do have some usability advantages, such as **error prevention** (the modal dialog is always on top, so it can't get lost or be ignored, and the user can't accidentally change the selection in the main window while working on a modal dialog that affects that selection).

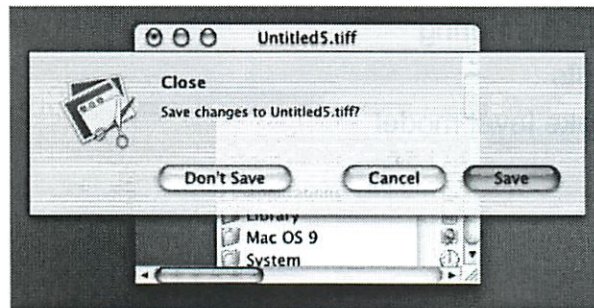
But there are usability disadvantages too, chief among them **loss of user control** and reduced **visibility** (e.g., you can't see important information or previews in the main window, and can't scroll the main window to bring something else into view). Modal dialogs may also overload the user's **short-term memory** – if the user needs some information from the main window, or worse, from a second modal dialog, then they're forced to remember it, rather than simply viewing and interacting with both dialogs side-by-side.

When you try to interact with the main window, Windows gives some nice animated **feedback** – flashing the border of the modal dialog box. This helps explain why your clicks on the main window had no effect.

On most platforms, you can at least move, resize, and minimize the main window, even when a modal dialog is showing. (The modal dialog minimizes along with it.) Alas, not on Windows... the main window is completely pinned! You can minimize it only by obscure means, like the Show Desktop command, which minimizes *all* windows. This is a big obstacle to user control and freedom.

Modeless dialogs, by contrast, don't prevent using other windows in the application. They're often used for ongoing interactions with the main window, like Find/Replace. One problem is that a modeless dialog box can get in the way of viewing or interacting with the main window (as when a Find/Replace dialog covers up the match). Another problem is a **consistency** problem: modal dialogs and modeless dialogs usually look identical. Sometimes the presence of a Minimize button is a clue that it's modeless, but that's not a very strong visual distinction. A modeless dialog may be better represented as a **sidebar**, a temporary pane in the main window that's anchored to one side of the window. Then it can't obscure the user's work, can't get lost, and is clearly visually different from a modal dialog box.

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

3

On Windows, modal dialogs are generally *application-modal* – all windows in the application stop responding until the dialog is dismissed. (The old days of GUIs also had *system-modal* dialogs, which suspended *all* applications.) Mac OS X has a neat improvement, *window-modal* dialogs, which are displayed as translucent sheets attached to the titlebar of the blocked window. This tightly associates the dialog with its window, gives a little visibility of what's underneath it in the main window – and allows you to interact with other windows, even if they're from the same application.

Another advantage of Mac sheets is that they make a strong contrast with modeless dialogs – the translucent, anchored modal sheet is easy to distinguish from a modeless window.

Today's Topics

- Chunking
- Pointing & steering
- Shortcuts
- Keystroke level model

CHUNKING

Spring 2012

6.813/6.831 User Interface Design and Implementation

7

Chunking

- “Chunk” is a unit of memory or perception
 - Depends both on presentation and on what you already know

Hard: M W B C R A L O A B I M B F I

Easier: MWB CRA LOA BIM BFI

Easiest: BMW RCA AOL IBM FBI



Spring 2012

6.813/6.831 User Interface Design and Implementation

8

The elements of perception and memory are called **chunks**. In one sense, chunks are defined symbols; in another sense, a chunk represents the activation of past experience. Our ability to form chunks in working memory depends strongly on how the information is presented: a sequence of individual letters tend to be chunked as letters, but a sequence of three-letter groups tend to be chunked as groups. It also depends on what we already know. If the three letter groups are well-known TLAs (three-letter acronyms) with well-established chunks in long-term memory, we are better able to retain them in working memory.

Chunking is illustrated well by a famous study of chess players. Novices and chess masters were asked to study chess board configurations and recreate them from memory. The novices could only remember the positions of a few pieces. Masters, on the other hand, could remember entire boards, but only when the pieces were arranged in *legal* configurations. When the pieces were arranged randomly, masters were no better than novices. The ability of a master to remember board configurations derives from their ability to chunk the board, recognizing patterns from their past experience of playing and studying games. (De Groot, A. D., *Thought and choice in chess*, 1965.)

Working Memory

- Working memory
 - Small: 7 ± 2 "chunks"
 - Short-lived: ~10 sec
 - Maintenance rehearsal fends off decay (but costs attention)

Spring 2012

6.813/6.831 User Interface Design and Implementation

9

Working memory is where you do your conscious thinking. The currently favored model in cognitive science holds that working memory is not actually a separate place in the brain, but rather a pattern of **activation** of elements in the long-term memory. A famous result is that the capacity of working memory is roughly 7 ± 2 things (technically called "chunks"). That's pretty small! Although working memory size can be increased by practice (if the user consciously applies mnemonic techniques that convert arbitrary data into more memorable chunks), it's not a good idea to expect the user to do that. A good interface won't put heavy demands on the user's working memory.

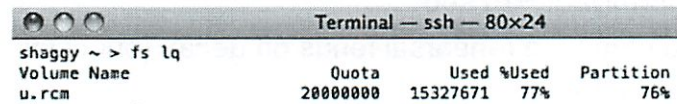
Data put in working memory disappears in a short time – a few seconds or tens of seconds. Maintenance rehearsal – repeating the items to yourself – fends off this decay, but maintenance rehearsal requires attention. So distractions can easily destroy working memory.

Long-term memory is probably the least understood part of human cognition. It contains the mass of our memories. Its capacity is huge, and it exhibits little decay. Long-term memories are apparently not intentionally erased; they just become inaccessible.

Maintenance rehearsal (repetition) appears to be useless for moving information into long-term memory. Instead, the mechanism seems to be **elaborative rehearsal**, which seeks to make connections with existing chunks. Elaborative rehearsal lies behind the power of mnemonic techniques like associating things you need to remember with familiar places, like rooms in your childhood home. But these techniques take hard work and attention on the part of the user. One key to good learnability is making the connections as easy as possible to make – and consistency is a good way to do that.

Example: Displaying Numbers

- Redesign this filesystem quota display so that it's more efficient



A terminal window titled "Terminal — ssh — 80x24" showing the output of the command "fs lq". The output is a table with four columns: Volume Name, Quota, Used %Used, and Partition. The data row shows "u.rcm" with a quota of 20000000, used space of 15327671 (77%), and partition 76%.

| Volume Name | Quota | Used %Used | Partition |
|-------------|----------|--------------|-----------|
| u.rcm | 20000000 | 15327671 77% | 76% |

Spring 2012

6.813/6.831 User Interface Design and Implementation

10

Let's use chunking to fix this information display.

While we're at it, what about learnability? What learnability problems does this UI have, and how might we fix them?

POINTING & STEERING

Spring 2012

6.813/6.831 User Interface Design and Implementation

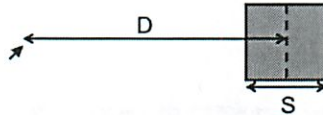
11

Fitts's Law

- Fitt's Law

- Time T to move your hand to a target of size S at distance D away is:

$$T = RT + MT = a + b \log(D/S + 1)$$

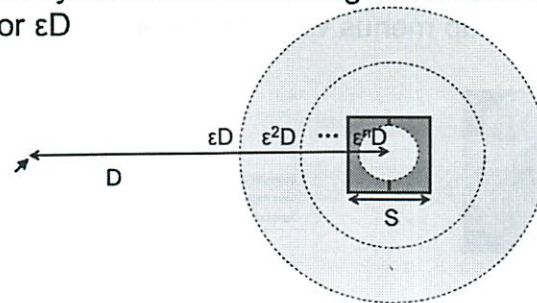


- Depends only on *index of difficulty* $\log(D/S + 1)$

Fitts's Law specifies how fast you can move your hand to a target of a certain size at a certain distance away (within arm's length, of course). 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. In the equation shown here, RT is reaction time, the time to get your hand moving, and MT is movement time, the time spent moving your hand.

Explanation of Fitts's Law

- Moving your hand to a target is closed-loop control
- Each cycle covers remaining distance D with error ϵD



Spring 2012

6.813/6.831 User Interface Design and Implementation

13

We can explain Fitts's Law by appealing to the human information processing model. Fitt's Law relies on closed-loop control. Assume that $D \gg S$, so your hand is initially far away from the target. 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 perceptual and cognitive processors perceive where your hand arrived and compare it to the target, and then your motor system issues a correction to move the remaining distance ϵD – which it does, but again with proportional error, so your hand is now within $\epsilon^2 D$. This process repeats, with the error decreasing geometrically, until n iterations have brought your hand within the target – i.e., $\epsilon^n D \leq S$. Solving for n , and letting the total time $T = n(T_p + T_c + T_m)$, we get:

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

where a is the reaction time for getting your hand moving, and $b = -(T_p + T_c + T_m)/\log \epsilon$.

The graphs above show the typical trajectory of a person's hand, demonstrating this correction cycle in action. The position-time graph shows an alternating sequence of movements and plateaus; each one corresponds to one cycle. The velocity-time graph shows the same effect, and emphasizes that hand velocity of each subsequent cycle is smaller, since the motor processor must achieve more precision on each iteration.

Implications of Fitts's Law

- Targets at screen edge are easy to hit
 - Mac menubar beats Windows menubar
 - Unclickable margins are foolish
- Linear popup menus vs. pie menus



Spring 2012

6.813/6.831 User Interface Design and Implementation

14

Fitts's Law has some interesting implications.

The edge of the screen stops the mouse pointer, so you don't need more than one correcting cycle to hit it. Essentially, the edge of the screen acts like a target with *infinite* size. (More precisely, the distance D to the center of the target is virtually equal to S , so $T = a + b \log(D/S + 1)$ solves to the minimum time $T=a$.) So edge-of-screen real estate is precious. The Macintosh menu bar, positioned at the top of the screen, is faster to use than a Windows menu bar (which, even when a window is maximized, is displaced by the title bar). Similarly, if you put controls at the edges of the screen, they should be active all the way to the edge to take advantage of this effect. Don't put an unclickable margin beside them.

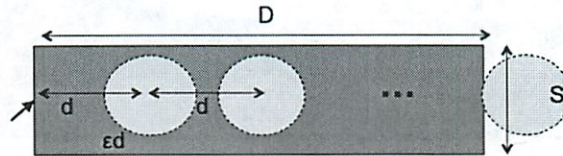
Fitts's Law also explains why pie menus are faster to use than linear popup menus. With a pie menu, every menu item is a slice of a pie centered on the mouse pointer. As a result, each menu item is the same distance D away from the mouse pointer, and its size S (in the radial direction) is comparable to D . Contrast that with a linear menu, where items further down the menu have larger D , and all items have a small S (height). According to one study, pie menus are 15-20% faster than linear menus (Callahan et al. "An empirical comparison of pie vs. linear menus," CHI 1991, <http://doi.acm.org/10.1145/57167.57182>). Pie menus are used occasionally in practice -- in some computer games, for example, and in the Sugar GUI created for the One-Laptop-Per-Child project. The picture here shows a pie menu for Firefox available as an extension. Pie menus are not widely used, however, perhaps because the efficiency benefits aren't large enough to overcome the external consistency and layout simplicity of linear menus.

Related to efficiency in general (though not to Fitts's Law) is the idea of a **gesture**, a particular movement of the mouse (or stylus or finger) that triggers a command. For example, swiping the mouse to the left might trigger the Back command in a web browser. Pie menus can help you learn gestures, when the same movement of your mouse is used for triggering the pie menu command (note that the Back icon is on the left of the pie menu shown). The combination of pie menus and gestures is called "marking menus", which have been used with good results in some research systems (Kurtenbach & Buxton, "User Learning and Performance with Marking Menus," CHI 1994. <http://www.billbuxton.com/MMUserLearn.html>)

Steering Tasks

- Time T to move your hand through a tunnel of length D and width S is:

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



- Index of difficulty is now *linear*, not logarithmic
 - So steering is much harder than pointing
- Thus cascading submenus are hard to use

Spring 2012

6.813/6.831 User Interface Design and Implementation

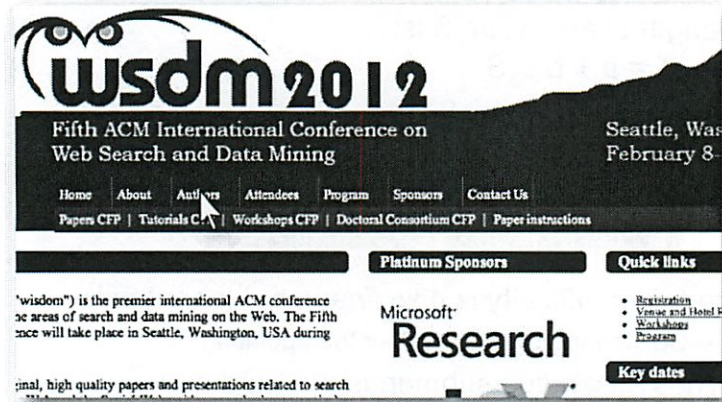
15

As we discussed in the first lecture, cascading submenus are hard to use, because the mouse pointer is constrained to a narrow tunnel in order to get over into the submenu. Unlike the pointing tasks that Fitts's Law applies to, this *steering task* puts a strong requirement on the error your hand is allowed to make: instead of iteratively reducing the error until it falls below the size of the target, you have to continuously keep the error smaller than the size of the tunnel. The figure shows an intuition for why this works. Each cycle of the motor system can only move a small distance d such that the error ϵd is kept below S . The total distance D therefore takes $D/d = \epsilon D/S$ cycles to cover. As a result, the time is proportional to D/S , not $\log D/S$. It takes *exponentially longer* to hit a menu item on a cascading submenu than it would if you weren't constrained to move down the tunnel to it.

Windows tries to solve this problem with a 500 ms timeout, and now we know another reason that this solution isn't ideal: it exceeds T_p (even for the slowest value of T_p), so it destroys perceptual fusion and our sense of causality. Intentionally moving the mouse down to the next menu results in a noticeable delay.

The Mac gets a Hall of Fame nod here: when a submenu opens, it provides an invisible 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. (Hall of Fame interfaces may sometimes be invisible to the user! They simply work better, and you don't notice why.)

Example: Steering Tasks on the Web



Spring 2012

6.813/6.831 User Interface Design and Implementation

16

Steering tasks are surprisingly common in systems with cascading submenus. Here's one on a website (<http://wsdm2012.org/>). Try hovering over Authors to see its submenu. How do you have to move the mouse in order to get to "Paper instructions"?

Improve Mouse Efficiency

- Make frequently-used targets big
 - Use snapping in drawing editors
- Put targets used together near each other
- Use screen corners and screen edges
- Avoid steering tasks

Spring 2012

6.813/6.831 User Interface Design and Implementation

17

Now that we've discussed aspects of the human cognitive system that are relevant to user interface efficiency, let's derive some practical rules for improving efficiency.

First, let's consider mouse tasks, which are governed by pointing (Fitts's Law) and steering. Since size matters for Fitts's Law, frequently-used mouse affordances should be big. The bigger the target, the easier the pointing task is.

Similarly, consider the path that the mouse must follow in a frequently-used procedure. If it has to bounce all over the screen, from the bottom of the window to the top of the window, or back and forth from one side of the window to the other, then the cost of all that mouse movement will add up, and reduce efficiency. Targets that are frequently used together should be placed near each other.

We mentioned the value of screen edges and screen corners, since they trap the mouse and act like infinite-size targets. There's no point in having an unclickable margin at the edge of the screen.

Finally, since steering tasks are so much slower than pointing tasks, avoid steering whenever possible. When you can't avoid it, minimize the steering distance. Cascading submenus are much worse when the menu items are long, forcing the mouse to move down a long tunnel before it can reach the submenu.

SHORTCUTS

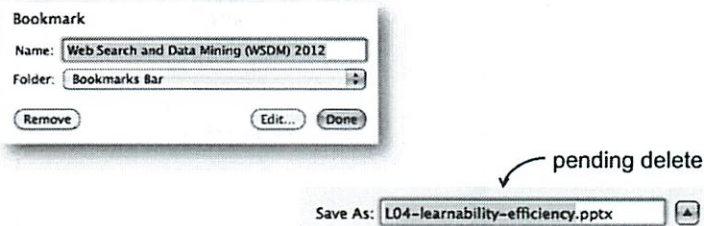
Spring 2012

6.813/6.831 User Interface Design and Implementation

18

Defaults & Pending Delete

- Fill in a form with defaults
 - from history, by prediction
- Make the defaults fragile



Spring 2012

6.813/6.831 User Interface Design and Implementation

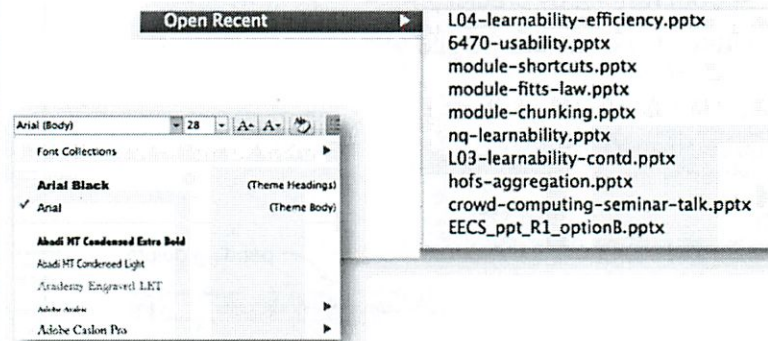
19

Defaults are common answers already filled into a form. Defaults help in lots of ways: they provide shortcuts to both novices and frequent users; and they help the user learn the interface by showing examples of legal entries. Defaults should be **fragile**; when you click on or Tab to a field containing a default value, it should be fully selected so that frequent users can replace it immediately by simply starting to type a new value. (This technique, where typing replaces the selection, is called **pending delete**. It's the way most GUIs work, but not all. Emacs, for example, doesn't use pending delete; when you highlight some text, and then start typing, it doesn't delete the highlighted text automatically.) If the default value is wrong, then using a fragile default allows the correct value to be entered as if the field were empty, so having the default costs nothing.

Incidentally, it's a good idea to remove the word "default" from your interface's vocabulary. It's a technical term with some very negative connotations in the lending world.

History

- Offer recently-used or frequently-used choices



Spring 2012

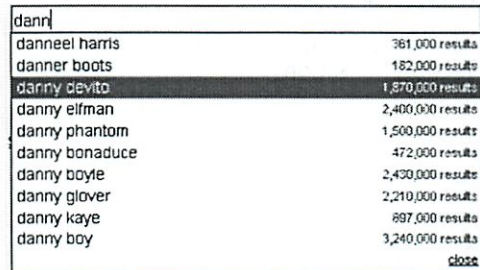
6.813/6.831 User Interface Design and Implementation

20

Many inputs exhibit temporal locality – i.e., the user is more likely to enter a value they entered recently. File editing often exhibits temporal locality, which is why Recently-Used Files menus (like this) are very helpful for making file opening more efficient. Keep histories of users' previous choices, not just of files but of any values that might be useful. When you display the Print dialog again, for example, remember and present as defaults the settings the user provided before.

Autocomplete

- Minimize typing with autocomplete



A screenshot of an autocomplete dropdown menu. The input field at the top contains the text 'dann'. Below it, a list of suggestions is shown, each followed by the number of results. The suggestion 'danny devito' is highlighted with a dark background. At the bottom right of the list is a 'close' button.

| | |
|----------------|-------------------|
| dann | |
| danneel harris | 361,000 results |
| danner boots | 182,000 results |
| danny devito | 1,870,000 results |
| danny elfman | 2,400,000 results |
| danny phantom | 1,500,000 results |
| danny bonaduce | 472,000 results |
| danny boye | 2,430,000 results |
| danny glover | 2,210,000 results |
| danny kaye | 897,000 results |
| danny boy | 3,240,000 results |
| | close |

Spring 2012

6.813/6.831 User Interface Design and Implementation

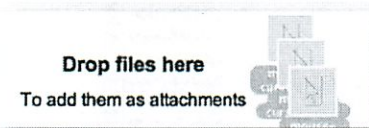
21

Autocomplete doesn't just help with efficiency. What other usability dimensions does it help?

Aggregation



multiple selection for action



multiple drag & drop

Spring 2012

6.813/6.831 User Interface Design and Implementation

22

Harkening back to the Hall of Fame & Shame for this lecture, **aggregation** is an excellent way to add efficiency to an interface. Think about ways that a user can collect a number of items – data objects, decisions, graphical objects, whatever – and handle them all at once, as a group. Multiple selection is a good design pattern for aggregation, and there are many idioms now for multiple selection with mouse and keyboard (dragging an outline around the items, shift-click to select a range, etc.)

Not every command needs aggregation, however. If the common case is only one item, and it's never more than a handful of items, then it may not be worth the complexity.

KEYSTROKE LEVEL MODELS

Spring 2012

6.813/6.831 User Interface Design and Implementation

23

Predictive Evaluation

- Predictive evaluation uses an engineering model of human cognition to predict usability
 - In this case, we'll predict efficiency
- The engineering model is
 - abstract
 - quantitative
 - approximate
 - estimated from user experiments

Spring 2012

6.813/6.831 User Interface Design and Implementation

24

Now we're going to turn to the question of how we can predict the efficiency of a user interface *before* we build it. Predictive evaluation is one of the holy grails of usability engineering. There's something worthy of envy in other branches of engineering -- even in computer systems and computer algorithm design -- in that they have techniques that can predict (to some degree of accuracy) the behavior of a system or algorithm before building it. Order-of-growth approximation in algorithms is one such technique. You can, by analysis, determine that one sorting algorithm takes $O(n \log n)$ time, while another takes $O(n^2)$ time, and decide between the algorithms on that basis. Predictive evaluation in user interfaces follows the same idea.

At its heart, any predictive evaluation technique requires a **model** for how a user interacts with an interface.

This model needs to be **abstract** -- it can't be as detailed as an actual human being (with billions of neurons, muscles, and sensory cells), because it wouldn't be practical to use for prediction.

It also has to be **quantitative**, i.e., assigning numerical parameters to each component. Without parameters, we won't be able to compute a prediction. We might still be able to do *qualitative* comparisons, such as we've already done to compare, say, Mac menu bars with Windows menu bars, or cascading submenus. But our goals for predictive evaluation are more ambitious.

These numerical parameters are necessarily **approximate**; first because the abstraction in the model aggregates over a rich variety of different conditions and tasks; and second because human beings exhibit large individual differences, sometimes up to a factor of 10 between the worst and the best. So the parameters we use will be averages, and we may want to take the variance of the parameters into account when we do calculations with the model.

Where do the parameters come from? They're estimated from experiments with real users. The numbers seen here for the general model of human information processing (e.g., cycle times of processors and capacities of memories) were inferred from a long literature of cognitive psychology experiments. But for more specific models, parameters may actually be estimated by setting up new experiments designed to measure just that parameter of the model.

Advantages of Predictive Evaluation

- Don't have to build UI prototype
 - Can compare design alternatives with no implementation whatsoever
- Don't have to test real live users
- Theory provides explanations of UI problems
 - So it points to the areas where design can be improved
 - User testing may only reveal problems, not explain them

Spring 2012

6.813/6.831 User Interface Design and Implementation

25

Predictive evaluation doesn't need real users (once the parameters of the model have been estimated, that is). Not only that, but predictive evaluation doesn't even need a **prototype**. Designs can be compared and evaluated without even producing design sketches or paper prototypes, let alone code.

Another key advantage is that the predictive evaluation not only identifies usability problems, but actually provides an **explanation** of them based on the theoretical model underlying the evaluation. So it's much better at pointing to *solutions* to the problems than either inspection techniques or user testing. User testing might show that design A is 25% slower than design B at a doing a particular task, but it won't explain *why*. Predictive evaluation breaks down the user's behavior into little pieces, so that you can actually point at the part of the task that was slower, and see why it was slower.

Keystroke-Level Model (KLM)

- **Keystroke**
- **Button press or release with mouse**
- **Point with mouse**
- **Draw line with mouse**
- **Home hands between mouse and keyboard**
- **Mentally prepare**

Spring 2012

6.813/6.831 User Interface Design and Implementation

26

The first predictive model was the **keystroke level model** (proposed by Card, Moran & Newell, “The Keystroke Level Model for User Performance Time with Interactive Systems”, *CACM*, v23 n7, July 1978).

This model seeks to predict efficiency (time taken by expert users doing routine tasks) by breaking down the user’s behavior into a sequence of the five primitive operators shown here.

Most of the operators are physical – the user is actually moving their muscles to perform them. The M operator is different – it’s purely mental (which is somewhat problematic, because it’s hard to observe and estimate). The M operator stands in for any mental operations that the user does. M operators separate the task into chunks, or steps, and represent the time needed for the user to recall the next step from long-term memory.

KLM Analysis

- Encode a method as a sequence of physical operators (KPHD)
- Use heuristic rules to insert mental operators (M)
- Add up times for each operator to get total time for method

Spring 2012

6.813/6.831 User Interface Design and Implementation

27

Here's how to create a keystroke level model for a task.

First, you have to focus on a particular **method** for doing the task. Suppose the task is deleting a word in a text editor. Most text editors offer a variety of methods for doing this, e.g.: (1) click and drag to select the word, then press the Del key; (2) click at the start and shift-click at the end to select the word, then press the Del key; (3) click at the start, then press the Del key N times; (4) double-click the word, then select the Edit/Delete menu command; etc.

Next, encode the method as a sequence of the physical operators: K for keystrokes, B for mouse button presses or releases, P for pointing tasks, H for moving the hand between mouse and keyboard, and D for drawing tasks.

Next, insert the mental preparation operators at the appropriate places, before each chunk in the task. Some heuristic rules have been proposed for finding these chunk boundaries.

Finally, using estimated times for each operator, add up all the times to get the total time to run the whole method.

Estimated Operator Times

- **Keystroke** determined by typing speed
 - 0.28 s average typist (40 wpm)
 - 0.08 s best typist (155 wpm)
 - 1.20 s worst typist
- **Button press or release**
 - 0.1 s highly practiced, no need to acquire button
- **Pointing** determined by Fitts's Law
$$T = a + b \log(d/s + 1) = a + b ID$$
 - 0.8 + 0.1 ID [Card 1978]
 - 0.1 + 0.4 ID [Epps 1986]
 - 0.1 + 0.2 ID [MacKenzie 1990, mouse selection]
 - 0.14 + 0.25 ID [MacKenzie 1990, mouse dragging]

OR

$T \sim 1.1$ s for all pointing tasks
- **Drawing** determined by steering law

Spring 2012

6.813/6.831 User Interface Design and Implementation

28

The operator times can be estimated in various ways.

Keystroke time can be approximated by typing speed. Second, if we use only an average estimate for K , we're ignoring the 10x individual differences in typing speed.

Button press time is approximately 100 milliseconds. Mouse buttons are faster than keystrokes because there are far fewer mouse buttons to choose from (reducing the user's reaction time) and they're right under the user's fingers (eliminating lateral movement time), so mouse buttons should be faster to press. Note that a mouse **click** is a press and a release, so it costs 0.2 seconds in this model.

Pointing time can be modelled by Fitts's Law, but now we'll actually need numerical parameters for it. Empirically, you get a better fit to measurements if the index of difficulty is $\log(D/S+1)$; but even then, differences in pointing devices and methods of measurement have produced wide variations in the parameters (some of them seen here). There's even a measurable difference between a relaxed hand (no mouse buttons pressed) and a tense hand (dragging). Also, using Fitts's Law depends on keeping detailed track of the location of the mouse pointer in the model, and the positions of targets on the screen. An abstract model like the keystroke level model dispenses with these details and just assumes that $T_p \sim 1.1$ s for all pointing tasks. If your design alternatives require more detailed modeling, however, you would want to use Fitts's Law more carefully.

Drawing time, likewise, can be modeled by the steering law: $T = a + b (D/S)$.

Estimated Operator Times

- **Homing** estimated by measurement
0.4 s (between keyboard and mouse)
- **Mental preparation** estimated by measurement
1.2 s

Spring 2012

6.813/6.831 User Interface Design and Implementation

29

Homing time is estimated by a simple experiment in which the user moves their hand back and forth from the keyboard to the mouse.

Finally we have the **Mental** operator. The M operator does not represent planning, problem solving, or deep thinking. None of that is modeled by the keystroke level model. M merely represents the time to prepare mentally for the next **step** in the method – primarily to retrieve that step (the thing you'll have to do) from long-term memory. A step is a chunk of the method, so the M operators divide the method into chunks.

The time for each M operator was estimated by modeling a variety of methods, measuring actual user time on those methods, and subtracting the time used for the physical operators – the result was the total mental time. This mental time was then divided by the number of chunks in the method. The resulting estimate (from the 1978 Card & Moran paper) was 1.35 sec – unfortunately large, larger than any single physical operator, so the number of M operators inserted in the model may have a significant effect on its overall time. (The standard deviation of M among individuals is estimated at 1.1 sec, so individual differences are sizeable too.) Kieras recommends using 1.2 sec based on more recent estimates.

Heuristic Rules for adding M's

- Basic idea:
 - M before every chunk in the method that must be recalled from long-term memory or that involves a decision
- Before each task or subtask
- Deciding which way to do a task
- Retrieving a chunk from memory
 - Command name
 - File name
 - Parameter value
- Finding something on screen
 - So P is often preceded by M
 - Unless the location is well-known from practice, in which case the visual search is overlapped with the motor action
- Verifying entry or action result
 - e.g. before pressing OK on a dialog

Spring 2012

6.813/6.831 User Interface Design and Implementation

30

One of the trickiest parts of keystroke-level modeling is figuring out where to insert the M's, because it's not always clear where the chunk boundaries are in the method. Here are some heuristic rules, suggested by Kieras ("Using the Keystroke-Level Model to Estimate Execution Times", 2001).

Example: Deleting a Word

- | | |
|---|---|
| <ul style="list-style-type: none">• Shift-click selection<ul style="list-style-type: none">MP [start of word]BB [click]MP [end of word]K [shift]BB [click]H [to keyboard]MK [Del]• Total: $3M + 2P + 4B + 1K$ = 6.93 sec | <ul style="list-style-type: none">• Del key N times<ul style="list-style-type: none">MP [start of word]BB [click]HMK [Del]x n [length of word]• Total: $2M + P + 2B + H + nK$ = $4.36 + 0.28n$ sec |
|---|---|

Spring 2012

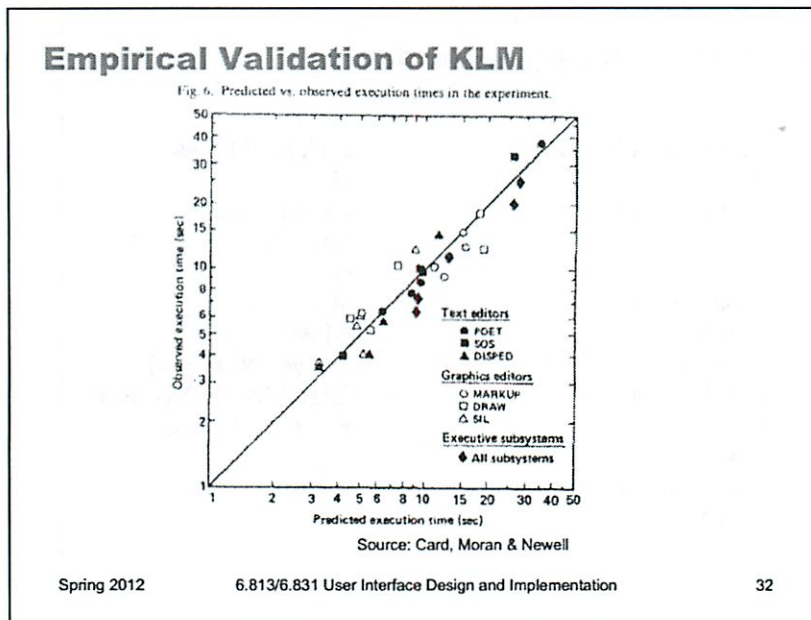
6.813/6.831 User Interface Design and Implementation

31

Here are keystroke-level models for two methods that delete a word.

The first method clicks at the start of the word, shift-clicks at the end of the word to highlight it, and then presses the Del key on the keyboard. Notice the H operator for moving the hand from the mouse to the keyboard. That operator may not be necessary if the user uses the hand already on the keyboard (which pressed Shift) to reach over and press Del.

The second method clicks at the start of the word, then presses Del enough times to delete all the characters in the word.



The developers of the KLM model tested it by comparing its predictions against the actual performance of users on 11 different interfaces (3 text editors, 3 graphical editors, and 5 command-line interfaces like FTP and chat).

28 expert users were used in the test (most of whom used only one interface, the one they were expert in).

The tasks were diverse but simple: e.g. substituting one word with another; moving a sentence to the end of a paragraph; adding a rectangle to a diagram; sending a file to another computer. Users were told the precise method to use for each task, and given a chance to practice the method before doing the timed tasks.

Each task was done 10 times, and the observed times are means of those tasks over all users.

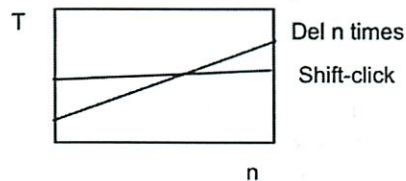
The results are pretty close – the predicted time for most tasks is within 20% of the actual time. (To give you some perspective, civil engineers usually expect that their analytical models will be within 20% error in at least 95% of cases, so KLM is getting close to that.)

One flaw in this study is the way they estimated the time for mental operators – it was estimated from the study data itself, rather than from separate, prior observations.

For more details, see the paper from which this figure was taken: Card, Moran & Newell, “The Keystroke Level Model for User Performance Time with Interactive Systems”, *CACM*, v23 n7, July 1978.

Applications of KLM

- Comparing designs & methods
- Parametric analysis



Spring 2012

6.813/6.831 User Interface Design and Implementation

33

Keystroke level models can be useful for comparing efficiency of different user interface designs, or of different methods using the same design.

One kind of comparison enabled by the model is **parametric analysis** – e.g., as we vary the parameter n (the length of the word to be deleted), how do the times for each method vary?

Using the approximations in our keystroke level model, the shift-click method is roughly constant, while the Del- n -times method is linear in n . So there will be some point n below which the Del key is the faster method, and above which Shift-click is the faster method. Predictive evaluation not only tells us that this point exists, but also gives us an estimate for n .

But here the limitations of our approximate models become evident. The shift-click method isn't really constant with n – as the word grows, the distance you have to move the mouse to click at the end of the word grows likewise. Our keystroke-level approximation hasn't accounted for that, since it assumes that all P operators take constant time. On the other hand, Fitts's Law says that the pointing time would grow at most logarithmically with n , while pressing Del n times clearly grows linearly. So the approximation may be fine in this case.

Limitations of KLM

- Only expert users doing routine (well-learned) tasks
- Only measures efficiency
 - Not learnability or safety
- Ignores
 - errors (methods must be error-free)
 - parallel action (shift-click)
 - mental workload (e.g. attention & WM limits)
 - planning & problem solving (how does user select the method?)
 - fatigue

Spring 2012

6.813/6.831 User Interface Design and Implementation

34

Keystroke level models have some limitations -- we've already discussed the focus on expert users and efficiency. But KLM also assumes no errors made in the execution of the method, which isn't true even for experts. Methods may differ not just in time to execute but also in propensity of errors, and KLM doesn't account for that.

KLM also assumes that all actions are serialized, even actions that involve different hands (like moving the mouse and pressing down the Shift key). Real experts don't behave that way; they overlap operations.

KLM also doesn't have a fine-grained model of mental operations. Planning, problem solving, different levels of working memory load can all affect time and error rate; KLM lumps them into the M operator.

Summary

- Chunking
- Pointing & steering
- Shortcuts
- Keystroke-level model

6.813/6.831 • USER INTERFACE DESIGN AND IMPLEMENTATION

Spring 2012 Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

GR1: PROJECT PROPOSAL AND ANALYSIS

Due at 11:59 pm on Sunday, February 26, 2012, by making a page on the class wiki.

The heart of this course is a semester-long project, in which you will design, implement, and evaluate a user interface. User interface design is an iterative process, so you will build your UI not just once, but three times, as successively higher-fidelity and more complete prototypes. In order to have time for these iterations, we need to get started on the project as early as possible.

Each project group should consist of 3 people. If you need help finding group members, see the Groups Wanted page on the class wiki.

Choosing a Problem

For your project, you will choose a problem faced by some user population, and then design and implement a user interface to address that problem. Note that the problem should come first; try to avoid having preconceived notions about the solution until you've at least understood the problem.

You have a lot of freedom in choosing your problem topic. Previous year's projects (which you'll find linked from the course wiki) may provide some inspiration for problems, but use this only as a starting point. We want you to reach for a problem that's less familiar. A conventional web app -- a database of type X (where X may be recipes, courses, books, ...) that lets users log in, create items of type X, and search and browse for other people's X items -- is unlikely to give you a deep experience of original user interface design. You have already seen and used dozens of websites like that; you know how it's done.

Instead, you will learn the most from this project if you stretch yourself into unfamiliar areas of the design space. Here are two ways to stretch (not the only ones):

Go mobile. Choose a problem that requires a mobile interface -- a web app or native app for a smartphone or tablet. Mobile interfaces impose significant design constraints -- screen size, necessary simplicity, touch interaction -- that will force you to keep it simple, think hard about what matters, and solve the problem in an original way. Also, mobile apps must be small, not sprawling, with only essential features, which make them a good fit for the low-fidelity prototyping techniques in this course.

Design for somebody else.. Choose a problem faced by a user population that nobody in your group belongs to. If you design for somebody much different from you, you will have to learn more about your users and their problems, you will have to challenge your assumptions, and you will have to be more creative in your designs. Examples include people with different roles (diving coaches, singers, mothers), different capabilities (children, elderly, people with disabilities), and different contexts (windsurfers, rock climbers, buskers). Note that your chosen user population must be reasonably available to you -- you will have to interview and observe them for this assignment, and you will have to do user testing with them later in the semester.

User Observation and Analysis

Start your design process by talking with at least 3 representative users who face the problem you are tackling. Observe them dealing with the problem in their real environment. When you write up your analysis, you must give us evidence that you interviewed and observed people, but don't provide a narrative of these sessions. Instead, offer your conclusions, and justify them when you can by referring to observations. For example, "grocery shoppers may be distracted by children; one mother was repeatedly harassed by her son to buy some candy." Also, don't identify the users you interviewed by name.

haha - I think about both at once...

lots of paperwork

my idea

might be too narrow

check

check w/ my idea

we can

Using your observations, write up the following:

- **User analysis.**

Identify the characteristics of your user population. If you have multiple user classes, describe each one.

- **Task analysis.**

Determine the tasks of the problem you've chosen, analyze their characteristics, and answer the general questions about tasks we asked in lecture. Think about other questions you should ask that might be relevant to your particular domain. You should find and analyze at least 3 high-level tasks. If you can't find 3 interesting tasks, then your problem may be too small to serve as a good project, and you should rethink it.

What to Hand In

Can have extra stuff - stats, etc

For this assignment, you need to create a page for your project on the class wiki. Each group assignment will add more information to your group's wiki page over the semester. By the end of the semester, your wiki page will constitute the final report for your project.

The wiki is accessible from the course web page. The homepage of the wiki has instructions for creating your project page and linking it from the wiki homepage.

Your project page should include the following parts:

- **Group members**

A list of your group members. Students who are not registered for credit are not permitted to participate in the group project. All members of your group must be registered for the class by the time this assignment is due.

- **Problem statement**

Briefly state the problem(s) that your project will seek to solve. Take the user's point of view. Consider what the user's goals are, and what obstacles lie in the way.

- **GR1 Analysis**

Write up your user analysis and task analysis clearly, concisely, and completely. Put your GR1 Analysis on a new page of the wiki, with a link to it from your main project page.

Feel free to use the wiki for working documents and communication within your group, but don't clutter your primary project page with this internal communication.

Questions to Ask About a Task

- Why is the task being done?
- What does the user need to know or have before doing the task?
- Where is the task performed?
 - At a kiosk, standing up
- What is the environment like? Noisy, dirty, dangerous?
 - Outside
- How often is the task performed?
 - Perhaps a couple times a day
- What are its time or resource constraints?
 - A minute or two (might be pressed for time!)
- How is the task learned?
 - By trying it
 - By watching others
 - Classroom training? (probably not)
- What can go wrong? (Exceptions, errors, emergencies)
 - Enter wrong country code
 - Enter wrong user name
 - Get distracted while recording message
- Who else is involved in the task?

Spring 2011

6.813/6.831 User Interface Design and Implementation

25

There are lots of questions you should ask about each task. Here are a few, with examples relevant to the OMS send-message task. Collecting this information about tasks helps inform your design.

MIT poster creator - staff (1) Done

MIT poster creator - student (X)

Student poster user (X)

Do everything: RSVP, email reminder

Web interface - Creator

Mobile interface - Viewer

Creators → detailed

Viewers → fast

Task analysis - new page

1. Create a Poster

2. Scan a poster + Add to Calendar

Methods 3. View RSVP

1. ILS

2. Webcal

3. Email

4. Gmail password

②

Poster - Yes

- occasionally

Make an entry in your iPhon

Go to calendar → new entry → alert

30 ~~10~~ sec

- time

- alert

- date

- title

Know QR codes

Yes - MIT Mobile app scans QR
2/month - primary

Would use if put entry in Calendar

If thought extra ~~info~~ info under code

Plaz; Test analysis

GR1 Mtg 2

~~GR1~~
2/26

Done

Move content

I will review

Name

~~Scan Sched~~

~~Scan Sched~~

Post it Pickit

ScanIt

RSVP It

ScanItAddIt

ScanAdd

RScanVP

RSVP It

Set up domain + SVN

G1813

02/27

L7 VT Soft. Arch

PS 1 out

Mall of Fame/shame Windows XP Alt + Tab

Low learnability - must see someone
tab consistent w/ other apps

Can send up in wrong app

Mouse affects Alt tab

Alt + Shift + Tab ergonomic issue

OS X Expose Overshoot easily
Optimized to go to last used window
- no muscle memory

F9 press once

Windows tile

Special icon on keyboard

4 finger down gesture

All windows arranged randomly

②

Nanoquiz

Pointing task - not constrained how you get there linear
Steering task - must go a certain way log

- Cascading submenus
- Scroll bar
 - i must stay within No
 - looks like it i
 - but it will snap after a certain max distance
 - edge of screen is ∞
 - So he considers Not
- touchpad/mouse - must reposition "clutch"
 - fits law breaks
- dragging files to trash
 - can overshoot/miss
 - so no

Autocompletion

efficiency - faster

Safety - can only pick on list when restricted
- but make broader mistake - if too fast
- or sees your browser history

③

Today: Starting implementation

- Design patterns
 - Approaches
 - in every GUI toolkit ever
 - WebUI
 - 1st did Java Swing
 - 2nd Flex
 - 3rd HTML + JS
 - only teach front end
 - 6.170 does rest
-

Design Patterns

View tree

each part of tree controls part of ~~whole~~ screen

Usually rectangular
"bounding box"

mutate tree to change output

redraw algorithms

attach listeners

- publish - subscribe
- event
- observer

auto layout algorithms

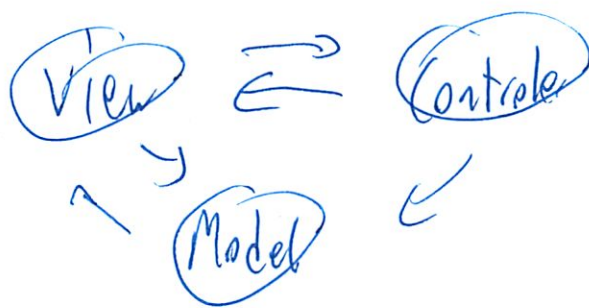
default objects have some
listener only focuses on some events

(4)

Model - View - Controller

- how we put it together
- how to separate UI from backend
 - ↳ the model
- won't be covered in this class

- so can reuse pieces
- easier to test + debug - hard to unit test
 - so can test model separately
- easier to give other pieces to other people
- (pattern chart from G.O.O.S)



(example of textbox in slides)

Just plug in your own model to JText Field
could have each keystroke sent to browser

5

Or a file ~~system~~ tree viewer example

Problems

Separating In & Out impossible

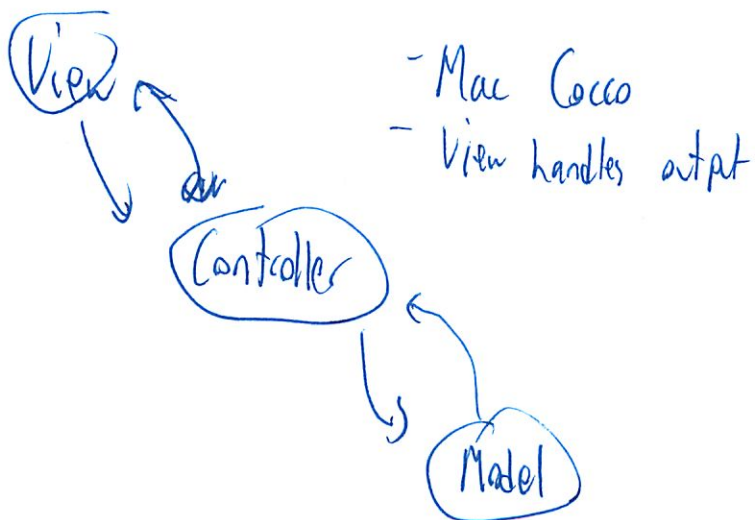
Controller needs to know a lot

So view must provide affordances for controller

View must provide feedback about controller state
eg scrollbar thumb
eg depressed buttons

So more like ~~View Controller Model~~
↳ Model View (MV)

Or change nature of controller



6

UI Approaches

- Procedural - how ~~represent~~ to get what you want
- declarative - says what you want
- Direct manipulation - ~~and~~ create it

Procedural

1. Put down block 1
2. " 2
3. " 3

Like Java Swing
or JS

procedural manipulation
of DOMs

HTML

↑ hard to do
browser input

(list of important HTML tags)

HTML imprinted on widgets

Declarative

I want tower
w/ 3
blocks

like HTML tags
- parser

Direct Manipulation



Or a UI builder

- make sure to get right
tree out of it

JS

var x = 5

↑ w/ global scope (w/o) var in local scope (Research more)

L7: UI Software Architecture

- PS1/RS1 out, due Sun
- make sure you're registered for the right course:
6.813 U or 6.831 G

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

2

Today's candidate for the Hall of Fame & Shame is the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

For those who haven't used it, here's how it works. Pressing Alt-Tab makes this window appear. As long as you hold down Alt, each press of Tab cycles to the next window in the sequence. Releasing the Alt key switches to the window that you selected.

The first observation to make is that this interface is designed only for keyboard interaction. Alt-Tab is the only way to make it appear; pressing Tab (or Shift-Tab) is the only way to cycle through the choices. If you try to click on this window with the mouse, it vanishes. The interface is weak on affordances, and gives the user little help in remembering how to use it.

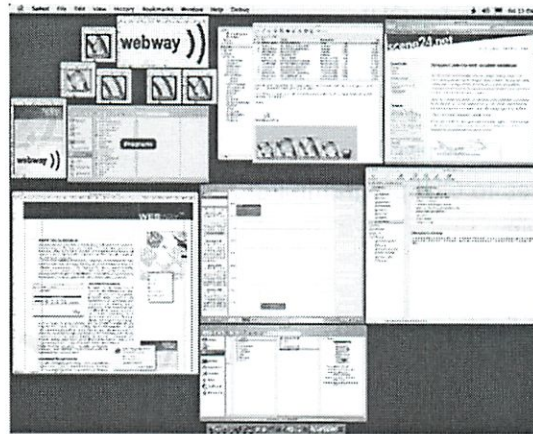
But that's OK, because the Windows taskbar is the primary interface for window switching, providing much better visibility and affordances. This Alt-Tab interface is designed as a **shortcut**, and we should evaluate it as such.

It's pleasantly **simple**, both in graphic design and in operation. Few graphical elements, good alignment, good balance. The 3D border around the window name could probably be omitted without any loss.

This interface is a **mode** (since pressing Tab is switching between windows rather than inserting tabs into text), but it's spring-loaded, happening only as long as the Alt button is held down.

Is it **efficient**? A common error, when you're tabbing quickly, is to overshoot your target window. You can fix that by cycling around again, but that's not as **reversible** as just moving backwards with a mouse. (You can also back up by holding down Shift when you press Tab, but that's not well-communicated by this interface, and it's tricky to negotiate while you're holding Alt down.)

UI Hall of Fame or Shame?



Spring 2012

6.813/6.831 User Interface Design and Implementation

3

For comparison, we'll also look at the Exposé feature in Mac OS X. When you push F9 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends the Exposé mode, sending all the other windows back to their old sizes and locations.

Like Alt-Tab, Exposé is also a **mode**. Unlike Alt-Tab, however, it is not spring-loaded. It depends instead on dramatic visual differences as a mode indicator – with its shrunken, tiled windows, Exposé mode usually looks a lot different than the normal desktop.

To get out of Exposé mode *without* choosing a new window, you can press F9 again, or you can click the window you were using before. That's easier to discover and remember than Alt-Tab's mechanism – pressing Escape. When I use Alt-Tab, and then decide to abort it, I often find myself cycling through all the windows trying to find my original window again. Both interfaces support **user control and freedom**, but Exposé seems to make canceling more **efficient**.

The representation of windows is much richer in Exposé than Alt-Tab (at least on Windows XP). Rather than Alt-Tab's icons (many of which are identical, when you have several documents open in the same application), Exposé uses the **window itself** as its visual representation. That's much more in the spirit of direct manipulation. (The version of Alt-Tab included in Windows Vista now shows images of the windows themselves – try it!)

Let's look at efficiency more deeply. Alt-Tab is a very linear interface – to pick an arbitrary window out of the n windows you have open, you have to press Tab $O(n)$ times. Exposé, on the other hand, depends on pointing – so because of Fitts's Law, the cost is more like $O(\log n)$. (Of course, this analysis only considers motor movement, not visual search time; it assumes you already know where the window you want is in each interface. But Exposé probably wins on visual search, too, since the visual representation shows the window itself, rather than a frequently-ambiguous icon.)

But Alt-Tab is designed to take advantage of **temporal locality**; the windows you visited recently are at the start of the list. So even if Exposé is faster at getting to an arbitrary window, Alt-Tab really wins on one very common operation: toggling back and forth between two windows.

Today's Topics

- Design patterns for GUIs
 - View tree
 - Listener
 - Widget
 - Model-view-controller
- Approaches to GUI programming
 - Procedural
 - Declarative
 - Direct manipulation
- Web UI at lightning speed
 - HTML
 - Javascript
 - jQuery

Spring 2012

6.813/6.831 User Interface Design and Implementation

6

Today's lecture is the first in the stream of lectures about how graphical user interfaces are implemented. Today we'll take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Three of the most important patterns are the **model-view-controller** abstraction, which has evolved somewhat since its original formulation in the early 80's; the **view tree**, which is a central feature in the architecture of every important GUI toolkit; and the **listener** pattern, which is essential to decoupling the model from the view and controller.

We'll also look at the three main approaches to implementing GUIs, and use that context for a quick introduction to HTML, Javascript, and jQuery, which together with CSS (next lecture) constitute the user interface toolkit that we'll be using in lectures and problem sets in this class. Note that the backend development of web applications falls outside the scope of the course material in this class. So we won't be talking about things like SQL, PHP, Ruby on Rails, or even AJAX. For more about that, you may want to check out the 6.470 IAP web programming competition, or the soon-to-be-offered 6.170 web programming software lab.

DESIGN PATTERNS

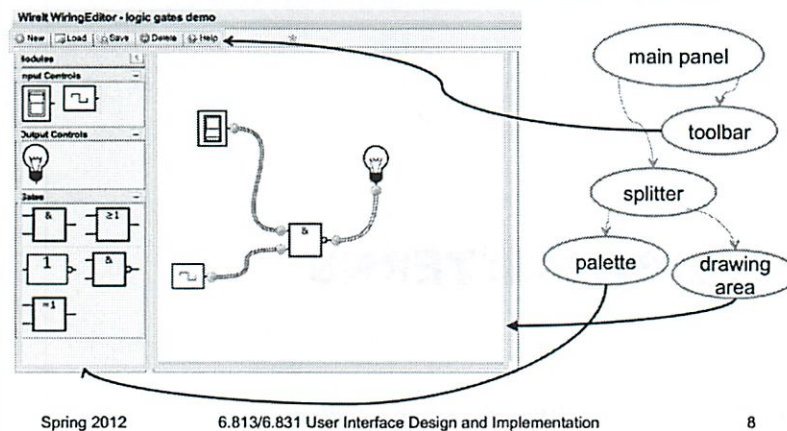
Spring 2012

6.813/6.831 User Interface Design and Implementation

7

View Tree

- A GUI is structured as a tree of views
 - A view is an object that displays itself on a region of the screen



Spring 2012

6.813/6.831 User Interface Design and Implementation

8

This leads to the first important pattern we'll talk about today: the **view tree**. A view is an object that covers a certain area of the screen, generally a rectangular area called its bounding box. The view concept goes by a variety of names in various UI toolkits. In Java Swing, they're `JComponents`; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.

Views are arranged into a hierarchy of containment, in which some views contain other views. Typical containers are windows, panels, and toolbars. The view tree is not just an arbitrary hierarchy, but is in fact a spatial one: child views are nested inside their parent's bounding box.

How the View Tree Is Used

- **Output**
 - GUIs change their output by **mutating** the view tree
 - A redraw algorithm automatically redraws the affected views
- **Input**
 - GUIs receive keyboard and mouse input by attaching listeners to views (more on this in a bit)
- **Layout**
 - Automatic layout algorithm traverses the tree to calculate positions and sizes of views

Spring 2012

6.813/6.831 User Interface Design and Implementation

9

Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

Output. Views are responsible for displaying themselves, and the view hierarchy directs the display process. GUIs change their output by mutating the view tree. For example, in the wiring diagram editor shown on the previous slide, the wiring diagram is changed by adding or removing objects from the subtree representing the drawing area. A redraw algorithm automatically redraws the affected parts of the subtree.

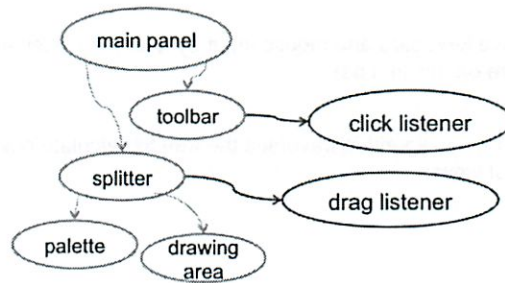
Input. Views can have input handlers, and the view tree controls how mouse and keyboard input is processed.

Layout. The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views.

We'll look at more about each of these areas in the next three lectures.

Input Handling

- Input handlers are associated with views
 - Also called **listeners**, event handlers, subscribers, observers



Spring 2012

6.813/6.831 User Interface Design and Implementation

10

- To handle mouse input, for example, we can attach a handler to the view that is called when the mouse is clicked on it. Handlers are variously called **listeners**, event handlers, subscribers, and observers.

Listener Pattern

- GUI input handling is an example of the Listener pattern
 - aka Publish-Subscribe, Event, Observer
- An event source generates a stream of discrete events
 - e.g., mouse events
- Listeners register interest in events from the source
 - Can often register only for specific events – e.g., only want mouse events occurring inside a view's bounds
 - Listeners can unsubscribe when they no longer want events
- When an event occurs, the event source distributes it to all interested listeners

Spring 2012

6.813/6.831 User Interface Design and Implementation

11

- GUI input event handling is an instance of the Listener pattern (also known as Observer and Publish-Subscribe). In the Listener pattern, an event source generates a stream of discrete events, which correspond to state transitions in the source. One or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs. In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an **event object** or passed as parameters.
- When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback functions.

MODEL-VIEW- CONTROLLER

Spring 2012

6.813/6.831 User Interface Design and Implementation

12

Separating Frontend from Backend

- We've seen how to separate input and output in GUIs
 - Output is represented by the view tree
 - Input is handled by listeners attached to views
- Missing piece is the backend of the system
 - Backend (aka **model**) represents the actual data that the user interface is showing and editing
 - Why do we want to separate this from the user interface?

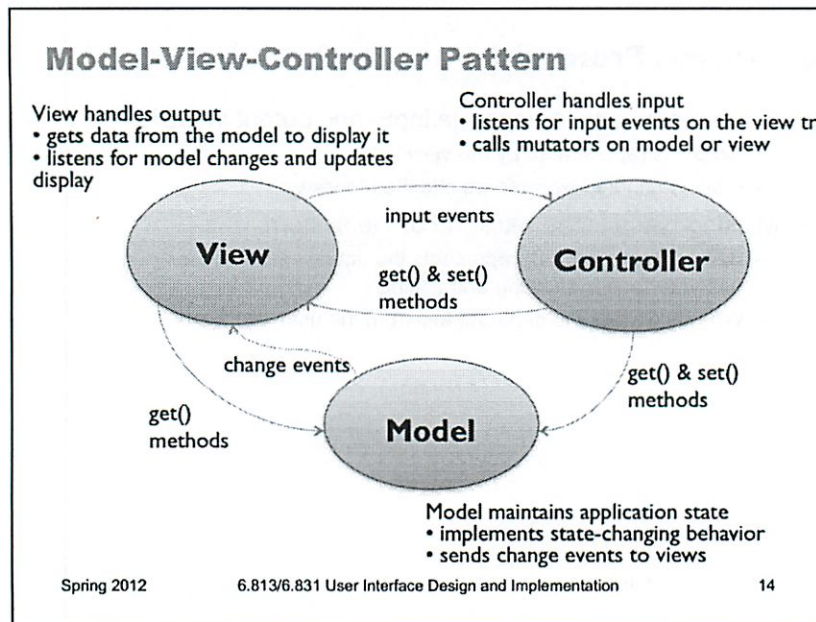
Spring 2012

6.813/6.831 User Interface Design and Implementation

13

We've seen how GUI programs are structured around a view tree, and how input events are handled by attaching listeners to views. This is the start of a separation of concerns – output handled by views, and input handled by listeners.

But we're still missing the application itself – the backend that actually provides the information to be displayed, and computes the input that is handled.



The **model-view-controller** pattern, originally articulated in the Smalltalk-80 user interface, has strongly influenced the design of UI software ever since. In fact, MVC may have single-handedly inspired the software design pattern movement; it figures strongly in the introductory chapter of the seminal “Gang of Four” book (Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Software*).

MVC’s primary goal is separation of concerns. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

The model is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately. Models do this notification using the **listener pattern**, in which interested views and controllers register themselves as listeners for change events generated by the model.

View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

Finally, the controller handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

Advantages of Model-View-Controller

- Separation of responsibilities
 - Each module is responsible for just one feature
 - Model: data
 - View: output
 - Controller: input
- Decoupling
 - View and model are decoupled from each other, so they can be changed independently
 - Model can be reused with other views
 - Multiple views can simultaneously share the same model
 - Views can be reused for other models, as long as the model implements an interface

Spring 2012

6.813/6.831 User Interface Design and Implementation

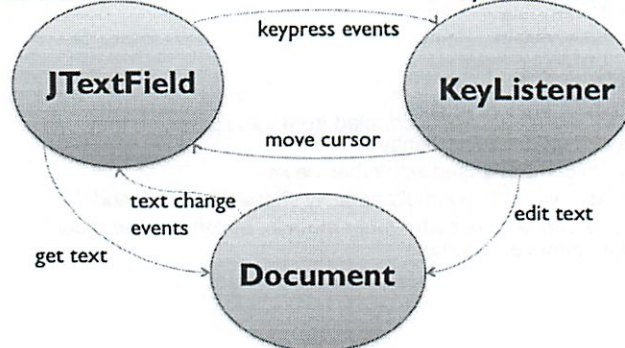
15

In principle, this separation has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable interface objects.

A Small MVC Example: Textbox

JTextField is a Component that can be added to a view tree

KeyListener is a listener for keyboard events



Document represents a mutable string of characters

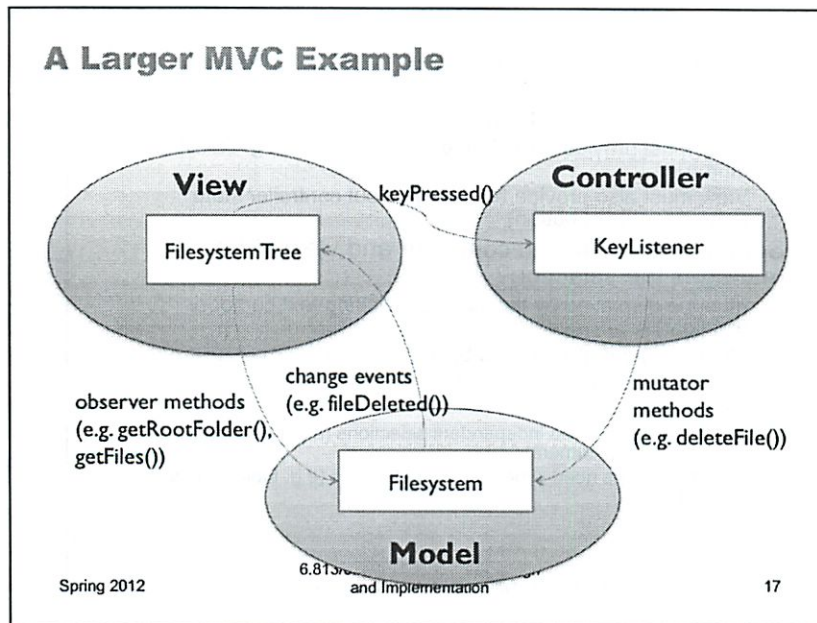
Spring 2012

6.813/6.831 User Interface Design and Implementation

16

A simple example of the MVC pattern is a text field widget (this is Java Swing's text widget). Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them in the string.

Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like the address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.



Here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.

Hard to Separate Controller and View

- Controller often needs output
 - View must provide **affordances** for controller (e.g. scrollbar thumb)
 - View must also provide **feedback** about controller state (e.g., depressed button)
- State shared between controller and view: Who manages the selection?
 - Must be displayed by the view (as blinking text cursor or highlight)
 - Must be updated and used by the controller
 - Should selection be in model?
 - Generally not
 - Some views need independent selections (e.g. two windows on the same document)
 - Other views need synchronized selections (e.g. table view & chart view)

Spring 2012

6.813/6.831 User Interface Design and Implementation

18

The MVC pattern has a few problems when you try to apply it, which boil down to this: you can't cleanly separate input and output in a graphical user interface. Let's look at a few reasons why.

First, a controller often needs to produce its own output. The view must display **affordances** for the controller, such as selection handles or scrollbar thumbs. The controller must be aware of the screen locations of these affordances. When the user starts manipulating, the view must modify its appearance to give **feedback** about the manipulation, e.g. painting a button as if it were depressed.

Second, some pieces of state in a user interface don't have an obvious home in the MVC pattern. One of those pieces is the **selection**. Many UI components have some kind of selection, indicating the parts of the interface that the user wants to use or modify. In our text box example, the selection is either an insertion point or a range of characters.

Which object in the MVC pattern should be responsible for storing and maintaining the selection? The view has to display it, e.g. by highlighting the corresponding characters in the text box. But the controller has to use it and modify it. Keystrokes are inserted into the text box at the location of the selection, and clicking or dragging the mouse or pressing arrow keys changes the selection.

Perhaps the selection should be in the model, like other data that's displayed by the view and modified by the controller? Probably not. Unlike model data, the selection is very transient, and belongs more to the frontend (which is supposed to be the domain of the view and the controller) than to the backend (the model's concern). Furthermore, multiple views of the same model may need independent selections. In Emacs, for example, you can edit the same file buffer in two different windows, each of which has a different cursor.

So we need a place to keep the selection, and similar bits of data representing the transient state of the user interface. It isn't clear where in the MVC pattern this kind of data should go.

Widget: Tightly Coupled View & Controller

- The MVC idea has largely been superseded by an MV (Model-View) idea
- A widget is a reusable view object that manages both its output and its input
 - Widgets are sometimes called components (Java, Flex) or controls (Windows)
- Examples: scrollbar, button, menubar

Spring 2012

6.813/6.831 User Interface Design and Implementation

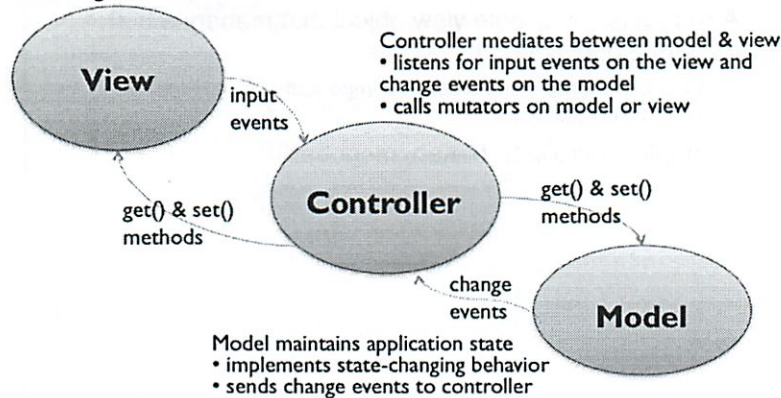
19

In principle, it's a nice idea to separate input and output into separate, reusable classes. In reality, it isn't always feasible, because input and output are tightly coupled in graphical user interfaces. As a result, the MVC pattern has largely been superseded by what might be called Model-View, in which the view and controllers are fused together into a single class, often called a **component** or a **widget**.

Most of the widgets in a GUI toolkit are fused view/controllers like this; you can't, for example, pull out the scrollbar's controller and reuse it in your own custom scrollbar. Internally, the scrollbar probably follows a model-view-controller architecture, but the view and controller aren't independently reusable.

A Different Perspective on MVC

View handles output & low-level input
• sends high-level events to the controller



Spring 2012

6.813/6.831 User Interface Design and Implementation

20

Partly in response to this difficulty, and also to provide a better decoupling between the model and the view, some definitions of the MVC pattern treat the controller less as an input handler and more as a **mediator** between the model and the view.

In this perspective, the view is responsible not only for output, but also for low-level input handling, so that it can handle the overlapping responsibilities like affordances and selections.

But listening to the model is no longer the view's responsibility. Instead, the controller listens to both the model and the view, passing changes back and forth. The events receiving high-level input events from the view, like selection-changed, button-activated, or textbox-changed, rather than low-level input device events).

The Mac Cocoa framework uses this approach to MVC.

GUI Implementation Approaches

- Procedural programming
 - Code that says *how* to get what you want (flow of control)
- Declarative programming
 - Code that says *what* you want (no explicit flow of control)
- Direct manipulation
 - Creating what you want in a direct manipulation interface

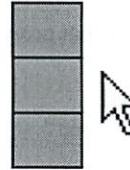
Procedural

1. Put down block A.
2. Put block B on block A.
3. Put block C on block B.

Declarative

A tower of 3 blocks.

Direct Manipulation



Spring 2012

6.813/6.831 User Interface Design and Implementation

21

Now let's talk about how to construct the view tree, which will be a tale of three paradigms.

In **procedural** style, the programmer has to say, step-by-step, how to reach the desired state. There's an explicit thread of control. This means you're writing code (in, say, Javascript or Java) that calls constructors to create view objects, sets properties of those objects, and then connects them together into a tree structure (by calling, say, `appendChild()` methods). Java Swing programming was largely procedural. Virtually every GUI toolkit offers an API like this for constructing and mutating the view tree.

In **declarative** style, the programmer writes code that directly represents the desired view tree. There are many ways to describe tree structure in textual syntax, but the general convention today is to use an HTML/XML-style markup language. There's no explicit flow of control in a declarative specification of a tree; it doesn't *do*, it just *is*. An automatic algorithm translates the declarative specification into runtime structure or behavior.

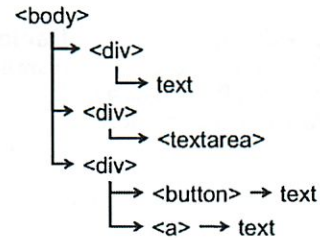
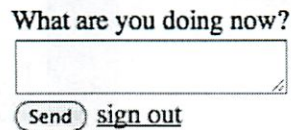
Finally, in **direct manipulation** style, the programmer uses a direct-manipulation graphical user interface to create the view tree. These interfaces are usually called GUI builders, and they offer a palette of view object classes, a drawing area to arrange them on, and a property editor for changing their properties.

All three paradigms have their uses, but the sweet spot for GUI programming basically lies in an appropriate mix of declarative and procedural – which is what HTML/Javascript provides.

Markup Languages

- HTML **declaratively** specifies a view tree

```
<body>
  <div>What are you doing now?</div>
  <div><textarea></textarea></div>
  <div><button>Send</button> <a href="#">sign out</a></div>
</body>
```



Spring 2012

6.813/6.831 User Interface Design and Implementation

22

Our first example of declarative UI programming is a **markup language**, such as HTML. A markup language provides a declarative specification of a view hierarchy. An HTML **element** is a component in the view hierarchy. The type of an element is its **tag**, such as `div`, `button`, and `img`. The properties of an element are its **attributes**. In the example here, you can see the `id` attribute (which gives a unique name to an element) and the `src` attribute (which gives the URL of an image to load in an `img` element); there are of course many others.

There's an automatic algorithm, built into every web browser, that constructs the view hierarchy from an HTML specification – it's simply an HTML parser, which matches up start tags with end tags, determines which elements are children of other elements, and constructs a tree of element objects as a result. So, in this case, the automatic algorithm for this declarative specification is pretty simple.

HTML Syntax

```
<!DOCTYPE html>
<html>
<head></head>
<body>
  <button disabled="true">
    <img src='dont.png' />
    Don't Press Me
  </button>
  <!--
                                <button>
                                  Press Me Instead
                                </button>
  -->
</body>
</html>
```

Spring 2012

6.813/6.831 User Interface Design and Implementation

23

Boilerplate: DOCTYPE, html, head, and body elements should be part of every HTML file.

An element consists of a start tag, attributes, content, and end tag.

Case doesn't matter for tag names and attribute names

Attribute values can be 'quoted' or "quoted"

- or not quoted at all, but it's better to quote

Text outside of a tag is grouped together into a "text node"

Whitespace is (mostly) ignored

Some kinds of elements are void (never have an end tag)

- e.g. img, br
- this is often reinforced with an extra slash: , both to help the reader, and because XML parsers demand it before they'll consider your HTML file (HTML is related to XML, and occasionally they try to play nicely together)

Comments look like <!-- -->

<, >, and & need to be escaped: < > & amp; respectively

Important HTML Elements for UI Design

- **Layout**
 - Box `<div>`
 - Grid `<table>, <tr>, <td>`
- **Text**
 - Font & color ``
- **Widgets**
 - Hyperlink `<a>`
 - Button `<button>`
 - Textbox `<input type="text">`
 - Multiline text `<textarea>`
 - Rich text `<div contenteditable="true">`
 - Drop-down `<select> <option>`
 - Listbox `<select multiple="true">`
 - Checkbox `<input type="checkbox">`
 - Radiobutton `<input type="radio">`
- **Pixel output**
 - ``
- **Stroke output**
 - `<canvas>` (Firefox, Safari)
- **Javascript code**
 - `<script>`
- **CSS style sheets**
 - `<style>`

Spring 2012

6.813/6.831 User Interface Design and Implementation

24

Here is a cheat sheet of the most important elements that you might use in an HTML-based user interface.

The `<div>` and `` elements are particularly important, and may be less familiar to people who have only used HTML for writing textual web pages. By default, these elements have no presentation associated with them; you have to add it using style rules (which we'll explain next lecture). The `<div>` element creates a box, and the `` element changes textual properties like font and color while allowing its contents to flow and word-wrap.

HTML has a rather limited set of widgets. There are other declarative UI languages similar to HTML that have much richer sets of built-in components, such as MXML (used in Adobe Flex) and XUL (used in Mozilla Firefox) and XAML (used in Microsoft WPF and Silverlight).

We'll talk more about the output elements, `img` and `canvas`, in the output lecture.

The `<script>` element to embed procedural code (usually Javascript) into an HTML specification. This actually breaks the model of declarative programming, because it introduces an explicit flow of control! The `<script>` elements are executed in the order that they are encountered in parsing the HTML, which means that they might see only a partially-constructed tree.

Finally, the `<style>` element is used for embedding another declarative specification, CSS style sheets, which we'll look at next lecture.

Exercise

- Use htmledit.squarefree.com to construct the UI shown below
 - you'll need `<div>`, `<textarea>`, `<button>`, `<a>`

What are you doing now?

[sign out](#)

- Use your browser's developer tools to inspect the UI you just created
 - e.g. find the `<textarea>`

View Tree Manipulation

- Javascript can **procedurally** mutate a view tree

```
<script>
var doc = document
var div1 = doc.createElement("div")
  div1.appendChild(doc.createTextNode("What are you doing now?"))
  ...
var div3 = doc.createElement("div")
var button = doc.createElement("button")
  button.appendChild(doc.createTextNode("Send"))
  div3.appendChild(button)
var a = doc.createElement("a")
  a.setAttribute("href", "#")
  a.appendChild(doc.createTextNode("sign out"))
  div3.appendChild(a)
</script>
```

What are you doing now?

 [sign out](#)

Spring 2012

6.813/6.831 User Interface Design and Implementation

26

Here's procedural code that generates the same HTML view tree, using Javascript and the Document Object Model (DOM). DOM is a standard set of classes and methods for interacting with a tree of HTML or XML objects procedurally. DOM interfaces exist not just in Javascript, which is the most common place to see it, but also in Java and other languages.

Note that the name DOM is rather unfortunate from our point of view. It has nothing to do with "models" in the sense of model-view-controller – in fact, the DOM is a tree of *views*. It's a model in the most generic sense we discussed in the Learnability lecture, a set of parts and interactions between them, that allows an HTML document to be treated as objects in an object-oriented programming language.

Most people ignore what DOM means, and just use the word (pronouncing it "Dom" as in "Dom DeLouise"). In fact DOM is often used to refer to the view tree.

Compare the procedural code here with the declarative code earlier.

Incidentally, you don't always have to use the `setAttribute` method to change attributes on HTML elements. In Javascript, many attributes are reflected as properties of the element (analogous to fields in Java). For example, `obj.setAttribute("id", value)` could also be written as `obj.id = value`. Be warned, however, that only standard HTML attributes are reflected as object properties (if you call `setAttribute` with your own wacky attribute name, it won't appear as a Javascript property), and sometimes the name of the attribute is different from the name of the property. For example, the "class" attribute must be written as `obj.className` when used as a property.

Raw DOM programming is painful, and worth avoiding. Instead, there are toolkits that substantially simplify procedural programming in HTML/Javascript -- jQuery is a good example, and the one we'll be using.

Javascript in One Slide

Like Java...

expressions

```
hyp = Math.sqrt(a*a + b*b)
console.log("Hello"
           + ", world");
```

statements

```
if (a < b) { return a }
else { return b }
```

comments

```
/* this
   is a comment */
// and so is this
```

Like Python...

no declared types

```
var x = 5;
for (var i = 0; i < 10; ++i) {...}
```

objects and arrays are dynamic

```
var obj = { x: 5, y: -1 };
obj.z = 8;
var list = ["a", "b"];
list[2] = "c";
```

functions are first-class

```
function square(x) { return x*x; }
var double = function(a) {
    return 2*a; }
```

Spring 2012

6.813/6.831 User Interface Design and Implementation

27

Here's everything you need to know about Javascript. Ha! Not exactly. But Javascript is not a hard language to pick up – it's a lot like Java and Python in many ways, and you probably already know Java and Python. Most of the differences are syntactic, which is visible and easy to learn by example. The trickiest pitfalls in Javascript (or in learning any language) are its semantics. Javascript's particular semantic pitfalls are **variable scoping** (which unlike Java is *function* scoped, not block scoped, and unlike Python it defaults to putting new variables in the *global* scope rather than the local scope) and the semantics of **this** (which doesn't behave quite like Java's this or Python's self). The variable scoping pitfalls are responsible for both warnings on this slide – (a) never omit the var keyword when you introduce a new variable, and (b) even though you should use var in your for loops, don't expect it to behave as in Java – there's only one variable i for the entire function, it isn't just scoped to the body of the for loop. A corollary of that is that functions you create within the body of the for loop all share the same variable i. (See "The Infamous Loop Problem" in <http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/>)

A good online tutorial for Javascript is "A re-introduction to JavaScript" (https://developer.mozilla.org/en/JavaScript/A_re-introduction_to_JavaScript).

Some good online articles describing the pitfalls of scoping and this:

- <http://jszen.blogspot.com/2005/04/variable-scoping-gotchas.html>
- <http://stackoverflow.com/questions/500431/javascript-variable-scope>
- <http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/>
- http://www.digital-web.com/articles/scope_in_javascript/

jQuery in One Slide

- Select nodes

| | |
|-----------------------------|---|
| <code>\$("#send")</code> | <code><button id="send" class="toolbar"></code> |
| <code>\$(".toolbar")</code> | <code>Send</code> |
| <code>\$("button")</code> | <code></button></code> |

- Create nodes

```
$('<button class="toolbar"></button>')
```

- Act on nodes

```
$("#send").text()           // returns "Send"  
$("#send").text("Tweet")   // changes button label  
$(".toolbar").attr("disabled", "true")  
$("#send").click(function() { ... })  
$("#textarea").val()  
$("#mainPanel").html("<button>Press Me</button>")
```

Spring 2012

6.813/6.831 User Interface Design and Implementation

28

jQuery offers a much better way to interact with the DOM than the actual DOM interface. jQuery is a Javascript library that you include in your HTML page. See jquery.com for more details, documentation, and tutorials.

The essence of jQuery is **selecting** a node (or set of nodes) in the DOM and **acting** on it (getting properties, setting properties, or changing tree structure).

Selection is done by a pattern language (which is a good pattern language to know because it's used in CSS as well, which we'll be learning about in the next lecture). For example, the pattern `#send` finds a node with the id attribute "send", `.toolbar` finds nodes with the class attribute "toolbar", and `button` just finds all `<button>` nodes.

jQuery provides a variety of methods for acting on the nodes you find. In general, jQuery methods come in pairs with the same name: the method with no arguments gets a value, and the method with arguments sets a value. So `.text()` returns the text contained in the node's descendants, while `.text("Tweet")` replaces all those descendants with the text node "Tweet". Similarly, `.attr()` gets and sets attribute values, `.click()` sets a mouse event handler (or simulates a click), `.val()` gets or sets the value of a text widget, and `.html()` gets or sets the descendants of a node as HTML.

Exercise

- Add jQuery to your user interface
`<script src="http://code.jquery.com/jquery-1.7.1.min.js"></script>`
- Attach an event listener to the Send button that displays the text area in your developer console
 - put id attributes on the Send button and the <textarea>
 - use \$("#id") to find an element by id
 - use .click() to attach an event handler
 - use .val() method to get the value of the textarea
 - use console.log() to print to the console

Mixing Declarative and Procedural Code

```
<body>
<div>What are you doing now?</div>
<div><textarea id="msg"></textarea></div>
<div><button id="send">Send</button></div>
<div id="sent" style="font-style: italic">
  <div>Sent messages appear here.</div>
</div>
</body>
```

What are you doing now?

Send

Sent messages appear here.

```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
```

```
<script>
```

```
$(function() {
  $("#send").click(function() {
    var msg = $("#msg").val()
  })
})
```

```
</script>
```

```
var sent = $("#sent").html()
sent += "<div>" + msg + "</div>"
$("#sent").html(sent)
```

```
var div =("<div></div>").text(msg)
$("#sent").append(div)
```

Spring 2012

6.813/6.831 User Interface Design and Implementation

30

To actually create a working interface, you frequently need to use a mix of declarative and procedural code. The declarative code is generally used to create the static parts of the interface, while the procedural code changes it dynamically in response to user input or model changes. Even inside the procedural code, we can use declarative code – a template of HTML that is filled with dynamically-computed parts.

One issue to think about is whether this template is constructed as a string of characters (as in the top green box), or as a data structure of objects (as in the bottom green box). Which do you think is better?

Note also that the code in the `<script>` tag is wrapped in a mysterious `$(function() {...})`, which is highlighted in red. This is jQuery shorthand for `$(document).ready(function() {...})`, which is in fact an event handler attached to the root of the view tree (the *document*). This event handler is called just once, after the entire HTML file has been parsed and the tree has been constructed. This is important to do! Why? Where could we put the `<script>` element so that the Send button doesn't even exist when the `<script>` element is executed? This is one of the ways that it's tricky to combine procedural and declarative programming.

Exercise

- Add a container for sent messages

```
<div id="sent" style="font-style: italic">  
  <div>Sent messages appear here.</div>  
</div>
```

- Change the Send button's click handler so that it adds messages to the sent container
 - use `.html()` to get or set the subtree under a node
 - use `.text()` to get or set the text inside a node
 - use `.append()` to add children to a node
 - use `$("<tag>...</tag>")` to create a subtree of HTML

Advantages & Disadvantages of Declarative UI

- Usually more compact
- Programmer only has to know how to say *what*, not *how*
 - Automatic algorithms are responsible for figuring out how
- May be harder to debug
 - Can't set breakpoints, single-step, print in a declarative specification
 - Debugging may be more trial-and-error
- Authoring tools are possible
 - Declarative spec can be loaded and saved by a tool; procedural specs generally can't

Spring 2012

6.813/6.831 User Interface Design and Implementation

32

Now that we've worked through our first simple example of declarative UI – HTML – let's consider some of the advantages and disadvantages.

First, the declarative code is usually more compact than procedural code that does the same thing. That's mainly because it's written at a higher level of abstraction: it says *what* should happen, rather than *how*.

But the higher level of abstraction can also make declarative code harder to debug. There's generally no notion of time, so you can't use techniques like breakpoints and print statements to understand what's going wrong. The automatic algorithm that translates the declarative code into working user interface may be complex and hard to control – i.e., small changes in the declarative specification may cause large changes in the output. Declarative specs need debugging tools that are customized for the specification, and that give insight into how the spec is being translated; without those tools, debugging becomes trial and error.

On the other hand, an advantage of declarative code is that it's much easier to build authoring tools for the code, like HTML editors or GUI builders, that allow the user interface to be constructed by direct manipulation rather than coding. It's much easier to load and save a declarative specification than a procedural specification. Some GUI builders *do* use procedural code as their file format – e.g., generating Java code and automatically inserting it into a class. Either the code generation is purely one-way (i.e., the GUI builder spits it out but can't read it back in again), or the procedural code is so highly stylized that it amounts to a declarative specification that just happens to use Java syntax. If the programmer edits the code, however, they may deviate from the stylization and break the GUI builder's ability to read it back in.

Summary

- Design patterns
 - View tree is the primary structuring pattern for GUIs, used for output, input, and layout
 - Listener is used for input and model-view communication
 - Model-view-controller decouples backend from GUI
- Approaches to GUI programming
 - Procedural, declarative, direct manipulation
 - HTML, Javascript, jQuery

PS1 Due Sun

Grp meetings w/ TA on Thur & Fri

GR2 out Mon, due next Sun - Grp will contact

Hall of Fame / Share: Windows XP Calc

- looks like a real calc
- #s changed like ~~num~~ num pad
- Multiplication sign is *
 - X in grade school
 - desktop calc X
 - but wanted consistency w/ keyboard
- sort = compat w/ programming
 - Who is your audience?
- not slavishly consistent w/ keyboard
- "Backspace" - like Windows keyboard
- Single line display
 - no history of current calc
 - or previous calc

(2)

Mode error

What # does depends on ^{hidden} mode

5 |

↑ append

then (+)

nothing happens to display

but next # clears display

and starts new #

→ Safety problem

Does not show entire # in memory

↳ could show whole #

Since big screen

Nanoquiz

~~Nanoquiz~~ Controller listens on view

not view listens to controller

~~nanoz mode~~

③

Mouse input should be handled by ~~view~~ controller
not view

Though often is handled by view (:

Declarative

- more compact
 - can be built w/ direct manipulation UI builder
-

Today's Error

- human error
 - capture
 - descriptions
 - modes
- prevention
- recovery
- messages
- tradeoffs: go faster → may be more error

④

Error types

Slip + Lapse

- failure to execute part of a procedure
- Slip → bad execution
- Lapse → failure of memory
- found in skilled behavior
 - ↳ you know what you are doing

Mistakes

Using wrong procedure for goal

Used in learning rule-based systems

Rule of thumb: Will mess up ~5% of time
↳ slip + lapse
most of the tasks

Mistakes ~50% of time

Since learning

~~fairly~~ fairly infreq - we mostly do existing procedures

⑤

Capture error

~~The first one~~

Two procedures start similar

- you pick wrong one when they diverge

Array formulas in Excel must be edited

by Ctrl + Shift - Enter

Whenever you edit it

Description error

- Perceive things wrong
- consistency good for learning
- but not inadvertent similarity
- like Wendall vs Kenmore St.
- 6.813 vs 6.831

6

Mode error

Same thing does different actions depending on mode

Must know mode you are in

- should be salient

Caps Lock & controlled several ways

VI & insert vs Command mode

Caps Lock not very visible - if not looking at screen

Or it diff place on each keyboard

Generally can't ~~compa~~ avoid modes

Best example: ^{universal} remote controls

Cause of slips

"Strong but wrong" effect

- Similarity - strong ~~que~~ cues from world

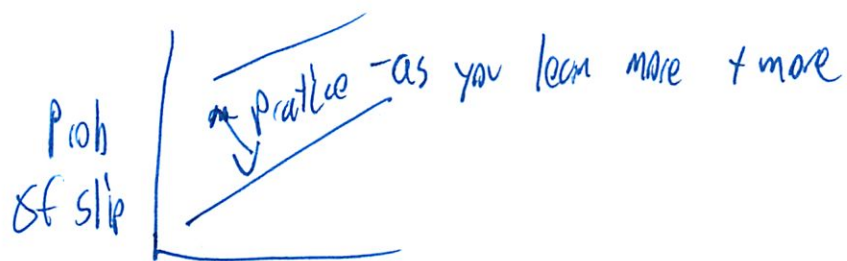
- high freq - muscle memory in jst hitting Enter
not Ctrl + Shift + Enter

- Or not paying attention

- You think you know it

②

Humans have a speed / accuracy trade off



Speed of execution

↑ linearly related w/ $\log(\text{slips})$

Error Prevention

Capture Errors

- Avoid action seq w/ same prefix

Description errors

- Analyze
- - similar names
- keep dangerous links away from common ones

Mode

- Eliminate modes
- Or make modes very clear
- Like which window are you in

⑧


Corners are horrible for visibility

- Mac OS X Bar
- Even worse when only on one ~~screen~~ screen

Or spring loaded

- you must actually hold down
- Shift
- Drag + Drop

Confirmation

- Common idiom for error prevention
- Can explain consequences more
- but 1 more button press
- i not work for slip
- People who are not ~~be~~ paying attention
 - hit Enter anyway
- Use sparingly
- undo better

①

Read User Freedom on own

Undo

More usable approach to confirmation

But hard to build mental model on how/what it actually undoes

- how many undo streams?
- units of undo?
- how much previous state is recovered?
 - selection
 - cursor position
- how visible is undo (if undone text is ~~not~~ hidden in a scroll box)?

Chrome: Global undo history - in page - but sep from address bar

FF: Each box has own undo stream

- depending on box that has active focus

Do sep frames have sep undo history?

Units -> Chrome aggregates. Unpredictable, whole group

FF - back to last non type char (like backspace) w/ small # of ^{chars}

(10)

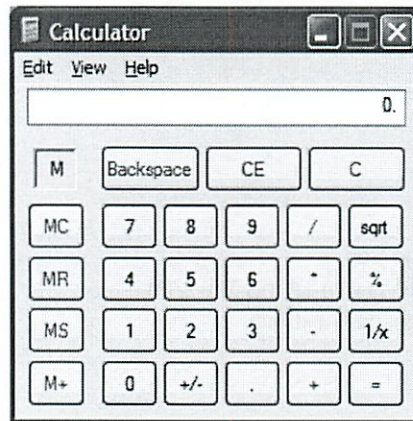
Undo: Chrome highlights undone text
but won't scroll to it if hidden

Cute idea in notes

L8: Safety

- PS1/RS1 due Sun
- Group meetings with TA this Thu & Fri
- GR2 out Mon, due next Sun

UI Hall of Fame or Shame?



Spring 2011

6.813/6.831 User Interface Design and Implementation

2

Today's candidate for the Halls of Fame and Shame is the Windows calculator.

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

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

Most of the buttons are cryptically worded (**recognition, not recall**). MC, MR, MS, and M+? What's the difference between CE and C? My first guess was that CE meant "Clear Error" (for divide-by-zero errors and the like); some people in class suggested that it means "Clear Everything". In fact, it means "Clear Entry", which just deletes the last number you entered without erasing the previous part of the computation. "C" actually clears everything.

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

Finally, we might also question the use of small blue text to label the buttons, which is hard to read, and the use of both red and blue labels in the same interface, since chromatic aberration forces red and blue to be focused differently. Both decisions tend to cause eyestrain over periods of long use.

Today's Topics

- Kinds of human error
 - capture, description, modes
- Error prevention
 - confirmation
- Error recovery
 - user control & freedom
 - undo
- Error messages

HUMAN ERROR

Spring 2012

6.813/6.831 User Interface Design and Implementation

6

Error Types

- Slips and lapses
 - Failure to correctly execute a procedure
 - Slip is a failure of execution, lapse is a failure of memory
 - Typically found in skilled behavior
- Mistakes
 - Using wrong procedure for the goal
 - Typically found in rule-based behavior or problem-solving behavior

Spring 2011

6.813/6.831 User Interface Design and Implementation

7

Errors can be classified into **slips and lapses** and **mistakes** according to how they occur.

Slips and lapses are found in skilled behavior – execution of procedures that the user has already learned. For example, pressing an onscreen button – moving the mouse pointer over it, pressing the mouse button, releasing the mouse button – is a skill-based procedure for virtually any computer user. An error in executing this procedure, like clicking before the mouse pointer is over the button, is a slip. This is just a low-level example, of course. We have many higher-level, learned procedures too – attaching a file to an email, submitting a search to Google, drawing a rectangle in a paint program, etc. An error in execution of any learned procedure would be a slip.

Slips are distinguished from lapses by the source of the failure. A slip is a failure of execution or control – for example, substituting one action for another one in the procedure. A lapse is a failure of memory – for example, forgetting the overall goal, or forgetting where you are in the procedure.

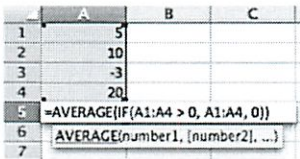
A **mistake**, on the other hand, is an error made in planning or rule application. One framework for classifying cognitive behavior divides behavior into skill-based (learned procedures), rule-based (application of learned if-then rules), and knowledge-based (problem solving, logic, experimentation, etc.) Mistakes are errors in rule-based or knowledge-based behavior; e.g., applying a rule in a situation where it shouldn't apply, or using faulty reasoning.

Overall, slips and lapses are more common than mistakes, because we spend most of our actual time executing learned procedures. If we spent most of our time problem-solving, we'd never get much done, because problem solving is such a slow, cognitively intensive, serial process. I've seen statistics that suggest that 60% of all errors are slips or lapses, but that's highly dependent on context. Relative to their task, however, slips and lapses are less common than mistakes. That is, the chance that you'll err executing any given step of a learned procedure is small -- typically 1-5%, although that's context dependent as well. The chance that you'll err in any given step of rule-based or problem-solving behavior is much higher.

We won't have much to say about mistakes in this lecture, but much research in human error is concerned with this level – e.g., suboptimal or even irrational heuristics that people use for decision making and planning. A great reference about this is James Reason, *Human Error*, Cambridge University Press, 1990.

Capture Errors

- Leave your house and find yourself walking to school instead of where you meant to go
- vi :w command (to save the file) vs. :wq command (to save and quit)
- Excel array formulas must be entered with Ctrl-Shift-Enter, not just Enter



| | A | B | C |
|---|-----------------------------------|---|---|
| 1 | 5 | | |
| 2 | 10 | | |
| 3 | -3 | | |
| 4 | 20 | | |
| 5 | =AVERAGE(IF(A1:A4 > 0, A1:A4, 0)) | | |
| 6 | =AVERAGE(number1, [number2], ...) | | |
| 7 | | | |

Spring 2011

6.813/6.831 User Interface Design and Implementation

8

Here are some examples of common slips. A **capture slip** occurs when a person starts executing one sequence of actions, but then veers off into another (usually 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 the text editor *vi*, it's common to quit the program by issuing the command ":wq", which saves the file (w) and quits (q). If a user intends just to save the file (:w) but accidentally quits as well (:wq), then they've committed a capture error. Microsoft Excel has a curious (and very useful!) class of formulas called array formulas, but in order to get Excel to treat your formula as an array formula, you have to press Ctrl-Shift-Enter after you type it – *every time* you edit it. Why is this prone to capture slips? Because virtually every other edit you do is terminated by Enter, so you're very likely to fall into that pattern automatically when you edit an array formula.

Description Errors



Kendall Station
Kenmore Station

6.813/6.831

| | | | |
|--------------------------|--------------------------|--------------------------|-------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Patrick Edward |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Mrs. Susan Iwan |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | 2012 coke award |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | USLUGI POSTA |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | BMC Software W |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Crisis Counseling |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Wine Masters Ch |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | April Lim |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | DotNetNuke Corp |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | ORC Internationa |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Ellen Gibson |

- Consistency (*same*) is good for learning
- Inadvertent similarity (*close-but-not-quite*) is bad for safety

Spring 2011

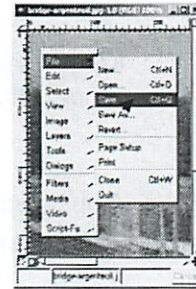
6.813/6.831 User Interface Design and Implementation

9

A **description slip** 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.

Mode Error

- Modes: states in which actions have different meanings
 - Vi's insert mode vs. command mode
 - Caps Lock
 - Drawing palette



Spring 2011

6.813/6.831 User Interface Design and Implementation

10

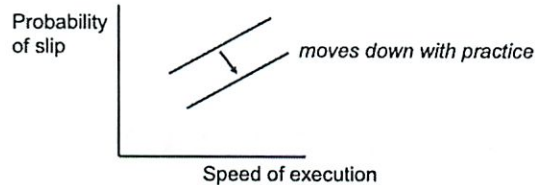
Another kind of error, clearly due to user interface, is a mode error. **Modes** are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands. In the first lecture, we talked about a mode error in Gimp: accidentally changing a menu shortcut because your mouse is hovering over it.

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.

Mode errors are generally slips, an error in the execution of a learned procedure, caused by failing to correctly evaluate the state of the interface.

Causes of Slips

- “Strong-but-wrong” effect
 - Similarity
 - High frequency
- Inattention or inappropriate attention
- Speed/accuracy tradeoff



Spring 2011

6.813/6.831 User Interface Design and Implementation

11

The slips and lapses we’ve discussed have a few features in common. First, the root cause of these errors is often **inattention**. Since slips and lapses occur in *skilled* behavior, execution of already well-learned procedures, they are generally associated with insufficient attention to the execution of the procedure, or omission or distraction of attention at a key moment.

Second, the particular erroneous behavior chosen is often selected because of its high similarity to the correct behavior (as in capture and description slips), or of its high frequency relative to the correct behavior (as in capture slips). Very common, or very similar, patterns are strongly available for retrieval from human memory. So errors are often **strong-but-wrong** behavior.

Finally, we can tune our performance to various points on a **speed-accuracy** tradeoff curve. We can force ourselves to make decisions faster (shorter reaction time) at the cost of making some of those decisions wrong. Conversely, we can slow down, take a longer time for each decision and improve accuracy. It turns out that for skill-based decision making, reaction time varies linearly with the log of odds of correctness; i.e., a constant increase in reaction time can double the odds of a correct decision.

The speed-accuracy curve isn’t fixed; it can be moved down and to the right by practicing the task. Also, people have different curves for different tasks; a pro tennis player will have a high curve for tennis but a low one for surgery.

One consequence of this idea is that **efficiency** can be traded off against **safety**. Most users will seek a speed that keeps slips to a low level, but doesn’t completely eliminate them.

ERROR PREVENTION

Spring 2012

6.813/6.831 User Interface Design and Implementation

12

Safety from Capture Errors

- Avoid habitual action sequences with identical prefixes

Spring 2011

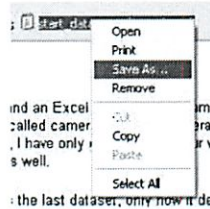
6.813/6.831 User Interface Design and Implementation

13

Let's discuss how to prevent errors of these sorts. In a computer interface, you can deal with capture errors by avoiding very common action sequences that have identical prefixes.

Safety from Description Errors

- Avoid actions with very similar descriptions
- Keep dangerous commands away from common ones



Spring 2011

6.813/6.831 User Interface Design and Implementation

14

Description errors can be fought off by applying the converse of the Consistency heuristic: different things should look and act different, so that it will be harder to make description errors between them. Avoid actions with very similar descriptions, like long rows of identical buttons.

You can also reduce description errors by making sure that dangerous functions (hard to recover from if invoked accidentally) are well-separated from frequently-used commands. Outlook 2003 makes this mistake: when you right-click on an email attachment, you get a menu that mixes common commands (Open, Save As) with less common and less recoverable ones – if you print that big file by mistake, you can't get the paper back. And if you Remove the attachment, it's even worse – undo won't bring it back! (Thanks to Amir Karger for this example.)

Safety from Mode Errors

- Eliminate modes
- Increase visibility of mode
- Spring-loaded or temporary modes
- Disjoint action sets in different modes

Spring 2011

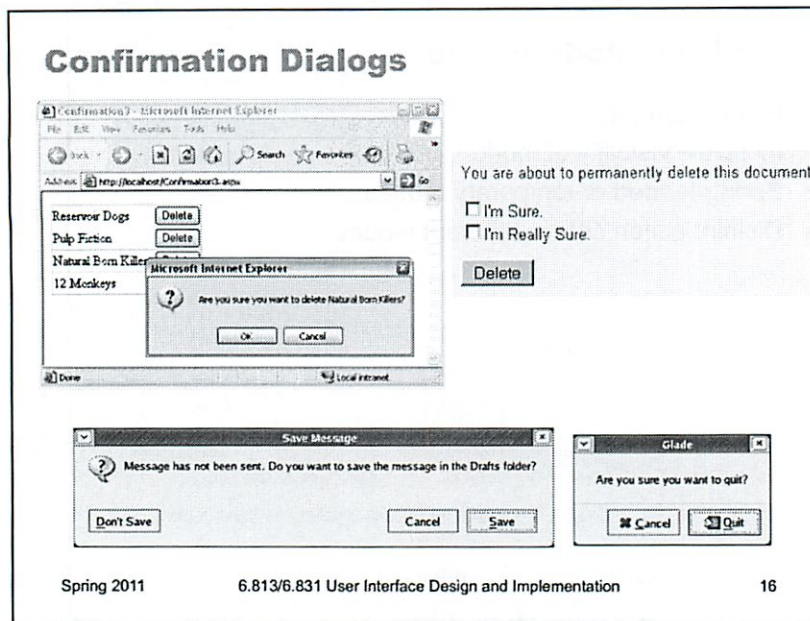
6.813/6.831 User Interface Design and Implementation

15

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. Modes do have *some* uses – they make command sets smaller, for example. 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.

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.



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 include it in their habitual procedure. In other words, to delete something, the user will learn to push Delete and then OK, without reading or even thinking about the confirmation dialog! The dialog has then completely lost its effectiveness, serving only to slow down the interface without actually preventing any errors.

In general, reversibility (i.e. **undo**) is a far better solution than confirmation. Even a web interface can provide at least single-level undo (undoing the last operation). Operations that are very hard to reverse may deserve confirmation, however. For example, quitting an application with unsaved work is hard to undo – but a well-designed application could make even this undoable, using automatic save or keeping unsaved drafts in a special directory.

USER CONTROL & FREEDOM

Spring 2012

6.813/6.831 User Interface Design and Implementation

17

User Control & Freedom

- Learning by exploring
- Dealing with errors
- User is sentient, computer is not

Spring 2011

6.813/6.831 User Interface Design and Implementation

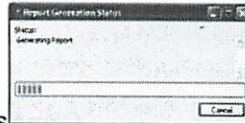
18

Good interfaces are **explorable**. One way users learn is by exploring: poking around an interface, trying things out. An interface should encourage this kind of exploration, not only by making things more visible, but also by making the consequences of errors less severe. For example, users navigating around a 3D world or a complex web site can easily get lost; give them an easy, obvious way to get back to some “home”, or default view. Users should be able to explore the interface without fear of being trapped in a corner.

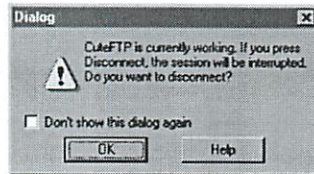
User control and freedom (a term coined by Jakob Nielsen) is the idea that in the give and take between the user and the system, the user should have ultimate control.

Clearly Marked Exits

- Long operations should be cancelable



- All dialogs should have a Cancel button



Source: Interface Hall of Shame

Spring 2011

6.813/6.831 User Interface Design and Implementation

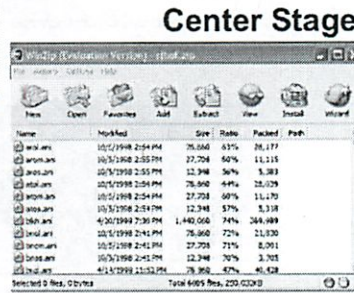
19

The simplest kind of user control is a veto – the ability to **cancel** an operation, even if it was something they asked for. Users should not be trapped by the interface. Long operations should not only have a progress bar, but a Cancel button too. Likewise, every dialog box should have a Cancel button. Where is it in this CuteFTP dialog box on the bottom? As a user of this dialog, would you feel like you're in control?

Wizard vs. Center Stage: Who's in Control?



Wizard



Center Stage

Spring 2011

6.813/6.831 User Interface Design and Implementation

20

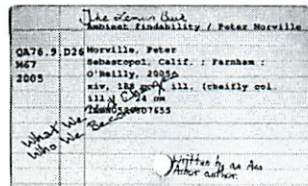
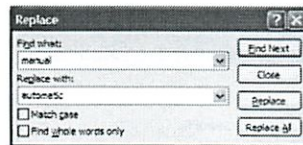
Let's look a little further at who controls the *dialog* between the user and the system. (Here, *dialog* means the general pattern of back-and-forth communication between the user and the interface, as if the user and the system are having a conversation. A *dialog box* is a specific kind of window, a design pattern used in a dialog. We often say *dialog* as a shorthand for *dialog box*, but hopefully the distinction will be obvious from context.)

We'll contrast two patterns. The **wizard** design pattern is a familiar pattern for improving the learnability of a complex interaction, by structuring it as a step-by-step process, with each step in a dialog. Wizards are the conventional pattern for software installation. In a wizard, the system controls the dialog – it dictates the steps, the ordering of the steps, and what it asks for at each step. Imagine a travel agent who's asking you a series of questions, and refuses to listen to what you say if it's not relevant to the question they asked. That's a wizard.

Contrast that with the **center stage** pattern, which lays out data objects in the main section of the window, and gives the user a set of tools for operating on the objects. In this case, the user controls the dialog, deciding which objects to select and which tools to pick up.

Wizards clearly restrict the user's freedom, but for complex, infrequently-done tasks (like installation), the tradeoff is often worth it. Note, however, that a good wizard has two key features: a Back button (for backing out of errors) and a Cancel button (for vetoing the operation entirely). So even though the wizard pattern puts the system in control of the details, the user still has **supervisory** control.

Manual Overrides for Automatic Systems



Source: www.findability.org



Spring 2011

6.813/6.831 User Interface Design and Implementation

21

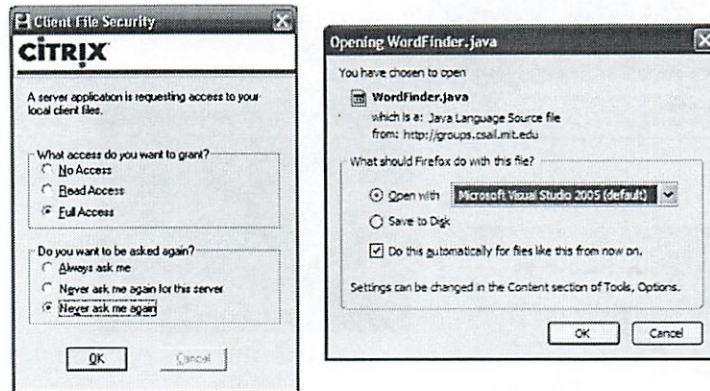
One of the main reasons we build software in the first place is to *automate* a process, taking some burden off the human users. But we can't take away control entirely. Users should be able to manually override automation.

The familiar Find & Replace command is a simple example of this. If Find & Replace were perfectly automatable, then all we'd need is Replace All. But the world isn't that simple, and our documents are full of exceptions or incompletely-specified patterns, and there are plenty of cases where the user needs manual control over replacement – hence the Find Next and Replace buttons.

Google Maps offers an example of a different kind of control – starting with the output of an automatic algorithm (the shortest route between two points) and manually tweaking it (dragging the route around). Systems that solve big or complex optimization problems should offer the user the opportunity to make these tweaks, since often there are constraints or preferences that are difficult to specify in advance, but can easily be seen when a solution is presented.

Some HCI researchers (prominently, Austin Henderson) argue that computer science in general, and corporate system developers in particular, have gone too far in trying to regularize the world, building systems that demand **coherence** from their users and their environment, expecting input that fits into expected categories and rejecting all others. For example, stating that every person has a first name and a last name, or assuming that every city belongs to only one country, or demanding a single shipping address for an order, are claims about the coherence of the world. But the real world is fuzzy, full of exceptions and oddities, and we should build **pliant systems** that can survive the exceptions. A great example of how paper-based systems are pliant is the **marginal comment**. Here's a card from an old-fashioned card catalog. You can easily distinguish the coherent typewritten data, which might fit neatly into a database system nowadays, from the marginalia. Margins on paper forms are often used by experienced workers to get their jobs done when the form is inadequate. We have a few design patterns for pliant user interfaces – such as comment fields (though they appear very rarely in business software!), and tagging instead of rigid hierarchies – but we don't really know how to build systems that are coherent enough for automation yet still pliant enough for the real world. (Jon Udell, "Scribbling in the Margins", Infoworld, http://www.infoworld.com/article/04/04/09/15OPstrategic_1.html)

Never Ask Me Again



Spring 2011

6.813/6.831 User Interface Design and Implementation

22

Here's an interesting problem related to who's in control of the dialog. Many interfaces interrupt users with questions, like the dialog boxes shown here. If the answer is always the same, it's clearly inefficient (and annoying) to keep asking the same question repeatedly – so many of these dialogs offer the option **Never ask me again**.

Good idea, and superficially seems to improve user control, because it's like a veto over all future questions of the same type. But suppose later the user wants to change their decision? Because the *system* initiated this dialog, not the *user*, the user has no idea how to return to the question. And the system has promised never to ask it again! It's a Catch-22.

One patch to this problem can be seen in the Firefox window on the right – a help message that tells the user where to look to undo the decision. But remember that just because the user has seen a message doesn't mean they've *learned* what it had to say. It's not clear that this really fixes the problem, but I haven't seen any better solutions.

User Control Over Data

- Data entered by the user should be editable by the user
- UI should give the power to:
 - Create a data item
 - Read it
 - Update it
 - Delete it

Spring 2011

6.813/6.831 User Interface Design and Implementation

23

So we've discussed user control over the dialog. Let's now consider user control over the data itself.

Editing is important. 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 the position of a rectangle – 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 user control and freedom.

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.

No Arbitrary Limits on User-Defined Names

→ The name contains too many capital letters.

Sign Up and Start Using Facebook

Join Facebook to **connect with your friends, share photos, and create your own profile**. Fill out the form below to get started (all fields are required to sign up).

Full Name: ←

I am:

Email:

Spring 2011

6.813/6.831 User Interface Design and Implementation

24

If an interface allows users to name things, then users should be free to choose long, descriptive names, with any characters or punctuation they want. Artificial limits on length or content should be avoided. DOS used to have a strong limit on filenames, an 8 character name and a 3 character extension, and a variety of punctuation characters are forbidden from filenames. Echoes of these limits persist in Windows even today.

Here's a bizarre requirement from Facebook (source: Error'd - The Daily WTF). No doubt the programmer's intention was to reject randomly-generated or nonsensical names which would reduce Facebook's appearance of professionalism, but the rule clearly doesn't work.

UNDO

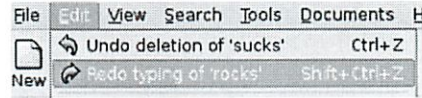
Spring 2012

6.813/6.831 User Interface Design and Implementation

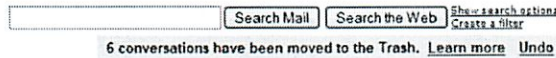
25

Support Undo

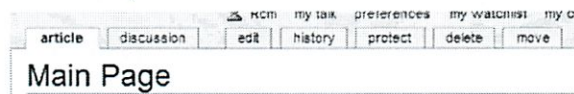
- Desktop



- Web



- Revision history



Spring 2011

6.813/6.831 User Interface Design and Implementation

26

If Cancel is the most common answer for user control over dialog, then Undo is the most common answer to user control over data. Undo has been around in desktop applications since the dark ages of the first Macintosh, if not before. The first Mac applications supported only **single-level undo** – that is, you could undo the last command, but no farther. This was largely due to memory constraints, and modern desktop applications allow unlimited undo (or so much that it makes no difference given the current interface for Undo – nobody is going to press Ctrl-Z 1000 times, after all).

Undo is also gradually appearing in web applications, like GMail. GMail's interface (shown here) only supports single undo. But other web applications support much longer undo histories, particularly apps designed for collaboration, like wikis. In these apps, undo typically takes the form of a revision history, rather than an undo command.

Forming a Mental Model of Undo

- Undo reverses the effect of an action
- But that leaves many questions:
 - What stream of actions will be undone?
 - How is the stream divided into undoable units?
 - Which actions are undoable, and which are skipped?
 - How much of the previous state is actually recovered by the undo?
 - How far back in the stream can you undo?

Spring 2011

6.813/6.831 User Interface Design and Implementation

27

You may think it's obvious what the Undo command does: it reverses the effect of the user's last action. But it's not as simple as that. Undo's behavior can be mysterious. Undo is an example of a case where the system model is not well communicated by the user interface. The actions managed by Undo are not visible; there's no persistent, visual representation showing the next action to be undone. (Not quite true: in well-designed interfaces, the Undo menu command's label gives a hint, like "Undo Typing" or "Undo Bold". But it's not prominent, so it doesn't particularly help a user form their mental model from ordinary use.) If you ask users to predict what effect Undo will have in some particular case, they may have no idea.

Let's look at some of the questions we should ask when we're designing an undo mechanism.

What stream of actions will be undone?

- Actions in this window (MS Office)
- Actions in this text widget (web browser)
- Just my actions, or everybody's (multiuser apps)
- Actions made by the computer
 - MS Office AutoCorrect and AutoFormat are undoable, even though user didn't do them

Spring 2011

6.813/6.831 User Interface Design and Implementation

28

Undo reverses the last action made by the user, but it's not necessarily the last one in the global stream. There is no global Undo in current GUI environments. Each application, sometimes even each widget, offers its own Undo command. A particular Undo command will only affect the action stream of the application or widget that it controls – so it will undo the last action in that application or widget's stream, which isn't necessarily the last command the user issued to the system as a whole.

Some applications use a separate action stream for each window. Microsoft Office works this way, for example. If you type something into Word document A, then type something else into Word document B, then switch back to A and invoke Undo, then A's insert will be undone – even though B's insert is the last one you actually performed.

Other applications treat each *text widget* as a separate action stream. Web browsers behave this way. Try visiting a form in a web browser, and type something into two different fields. You'll find that Undo only affects the field with the current keyboard focus, ignoring actions you made on any other fields. Changes made in other kinds of form widgets – drop-down menus or listboxes, for example – aren't added to *any* action stream.

Applications with multiple simultaneous users – such as a shared network whiteboard, where anybody can scribble on it – face the question of whether Undo should affect only your own actions, or everybody's actions. Usually, the best answer to this question is only your own actions, unless you have some kind of floor control mechanism that prevents people from working simultaneously [Abowd & Dix, "Giving undo attention," *Interacting with Computers*, v4 n3, 1992].

How is the stream divided into units?

- Lexical level
 - Mouse clicks, key presses, mouse moves
 - Nobody does it at this level
- Syntactic level
 - Commands and button presses
- Semantic level
 - Changes to application data structures (e.g., the result of an entire Format dialog)
 - This is the normal level
- Text entry is aggregated into a single action
 - But other editing commands (like Backspace) and newlines interrupt the aggregation
- What about user-defined macros?
 - Undo macro actions individually, or as a unit?

Spring 2011

6.813/6.831 User Interface Design and Implementation

29

Once you've decided which stream of actions to undo, the next question is, how is the stream divided into units? This is important because Undo reverses the last unit action of the stream.

Dividing at the **lexical level** means low-level input events, so Undo might reverse the very last keyboard or mouse change. For example, if you just did a drag-and-drop, invoking Undo might undo your mouse button release, putting you back into drag-and-drop mode and allowing you to drop somewhere else. No user interface (that I know of) implements lexical Undo in a systematic way; it's not clear how to get it right (since you're not holding the button down anymore!), and it's probably not what users want.

At the **syntactic level**, you would undo commands or onscreen button presses. For menu items and toolbar buttons, this is the right thing. But if you just finished a dialog – say, using the Font dialog, or selecting a Color – then this would undo the OK button press, returning you into the dialog box. Most applications don't do it at this level either.

The **semantic level** is what most designers choose, where Undo reverses the most recent change to the backend model – whether it was caused by a simple command, like Boldface, or a complicated dialog, like Page Layout. That's great for one kind of user control and freedom, since it makes complex changes just as easy to back out of as simple changes. But what if you just completed a long wizard dialog, only to discover that it didn't do what you wanted, and Undo only reverses the effect of the *entire* dialog, instead of getting you back into the wizard and letting you Back up? There are tradeoffs in the decision to undo only at the semantic level, but it's the most common.

For undoing text, individual typed characters should be **aggregated** somehow – otherwise, Undo won't be any faster than pressing Backspace. One natural way to do this might be word boundaries; but most text editors use edit commands and newlines as boundaries.

In general, the action stream should be divided into **chunks** from the user's perspective. For example, a user-defined macro is a chunk, so Undo should treat the entire macro as a unit action.

Which actions are undoable?

- User's action stream may include many actions that are ignored by Undo
 - Selection
 - Keyboard focus
 - Changing viewpoint (scrolling, zooming)
 - Changing layout (opening palettes or sidebars, adjusting window sizes)
 - UI customization (adding buttons to toolbars)
- So which actions does Undo actually undo?
 - Some applications (e.g. web browsers, IDEs) have Undo/Redo for the editing stream, Back/Forward for the viewpoint stream

Spring 2011

6.813/6.831 User Interface Design and Implementation

30

Many actions that affect visible program state may be completely ignored by Undo. Typically these actions affect the **view**, but don't actually change the backend model. Examples include selection, keyboard focus, scrolling and zooming, window management, and user interface customizations.

Since easy reversibility can be just as helpful for view changes, some applications define new commands for them, so they can reserve Undo for reversing model changes. Web browsers are a fine example: the Back button reverses a jump in view (whether caused by loading a new page or clicking on an internal hyperlink to jump to another place in the same page). Development environments like Eclipse have borrowed this idiom for navigation in code editors; you can press Back to undo window switching and scrolling.

How much state is recovered?

- Select text, delete it, and then undo
 - Text is restored
 - But is selection restored? Cursor position?

Spring 2011

6.813/6.831 User Interface Design and Implementation

31

Even if the Undo stream doesn't include all the view changes you make, how much of the view state will be restored when it reverses a model change? When you undo a text edit, for example, will the selection highlight be restored as well? Will the text cursor be put back where it was before the edit? If the text scrolls, will it be scrolled back to the same place?

How far back can you undo?

- Often a limit on history size
 - Used to be one action -- now usually hundreds, or infinite
- Does action stream persist across application sessions?
 - If so, stream must be saved to file
- Does it persist across File/Save?

Spring 2011

6.813/6.831 User Interface Design and Implementation

32

Finally, how far back will the undo history stream go? Old Macintosh applications had only single undo – i.e., you could only undo the last action, and no farther. Thankfully, cheap memory has made deep undo history feasible and commonplace.

Even though memory no longer limits undo, the conventional model of undo still does. In most applications, Undo is a transient phenomenon, limited to a single application session. If you shut down the application, and then restart it, the undo history is erased. So you can't undo past the start of the current session.

Some applications even erase the undo history as soon as the user saves a document to disk. Older versions of Microsoft Office used to behave this way.

Exercise: Undo Models

- Go to this web page
 - see Stellar, or shoutkey URL
- Explore your browser's undo model for textboxes
 - how many undo streams?
 - what are the units of undo?
 - how much previous state is recovered? (selections? cursor positions?)
 - how visible is the undo? (e.g., if the affected place is scrolled out of view of the browser page or textbox?)

one line, plain text

Purple Cow by Gertrude Burgess (published in The Lark, 1895)

multi-line, plain text

I never saw a Purple Cow,
I never hope to see one,
But I can tell you, anyhow,
I'd rather see than be one.

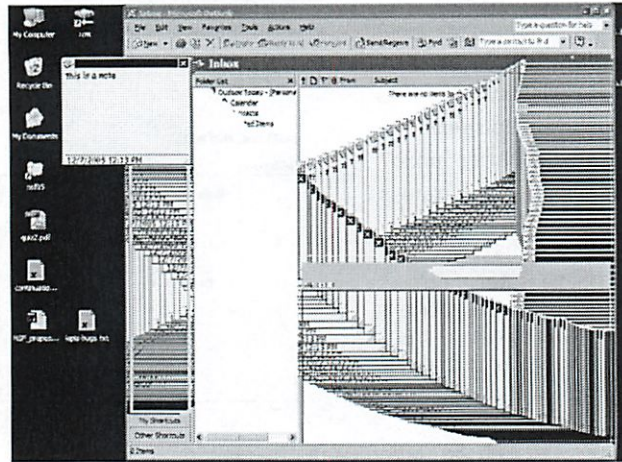
multi-line, rich text

Bold **Italic**

Reply (To The "Purple Cow")
by Gilbert Burgess
Apr 1914

Ah, yes, I wrote the "Purple Cow"~
I'm sorry, zow, I wrote it!
But I can tell you anyhow,
I'll kill you if you quote it.

Curious Case Study: Outlook Sticky Notes



Suggested by Chris Child

Spring 2011

6.813/6.831 User Interface Design and Implementation

34

Try this in Outlook 2007 (or Outlook 2003, but doesn't work in Outlook Express). Create a sticky note (File/New/Note). Type some text into the note, and move the note to a different place on the screen. Then press Ctrl-Z to undo. It undoes not only what you typed, but also the position of the note – and the note animates through all the different positions you moved it to on the screen.

Recall the important dimensions of an undo model:

- what stream of actions is undone? Only the actions that affected this sticky note; other sticky notes, and other Outlook windows, aren't affected.
- how is the stream divided into units? It turns out that the entire stream of actions since the note was created is a single unit – everything gets undone when you press Ctrl-Z once.
- what state is actually restored? everything about the note – its position, its size, even its color.
- how far back can you undo? As far as the creation of the note – unless you switch to another window. Switching away from the note clears the note's undo history, so further undo is impossible.

What else is wrong here? As the screenshot shows, the animation wasn't even done properly – instead of animating using automatic redraw, Outlook paints the moving note directly on the screen, leaving a smear behind it. Notice that the smear is visible in some parts of the Outlook window, but not in others. Why do you think that is?

Design Principles for Undo

- **Visibility**
 - Make sure undone effects are visible
 - e.g., scrolled into view, selected, possibly animated
- **Aggregation**
 - Units should be “chunks” of action stream: typed strings, dialogs, macros
- **Reversibility of the Undo itself**
 - Support Redo as well as Undo
 - Undo to a state where user can immediately reissue the undone command, or a variant on it
 - e.g., restore selection & cursor position
- **Reserve it for model changes, not view changes**
 - For consistency with other applications, reserve Undo for changes to backend data
- **“Undo” is not the only way to support reversibility**
 - Backspace undoes typing, Back undoes browsing, Recent Files undoes file closing, scrolling back undoes scrolling
 - Forward error recovery: using new actions to fix errors

Spring 2011

6.813/6.831 User Interface Design and Implementation

35

The upshot of all these questions is that it's very hard for users to predict what Undo will do. Faced with this unpredictability, a common strategy is to press Undo until you see the effect you want to reverse actually go away, or until you realize it's gone too far without solving the problem (i.e., it's reversed an older, still-desired effect). So **visibility** of Undo's effects is a critical part of making it usable. Whenever Undo undoes a command, it should make sure that the effects of that have a visible change on the screen. If the user has changed the viewpoint (e.g. scrolling) since doing the command that is now being undone, the viewpoint should be changed back, so that it's easy to see what was reversed.

The unit actions should correspond to **chunks** of the user's interaction: whole typed words (or strings), complete dialogs, user-defined macros.

Undo itself should be reversible, so that if you overshoot, you can come back. That's what the **Redo** command is for. Another way to reverse an Undo is to manually issue the undone command again; a good undo mechanism should set up the conditions for this as well. For example, suppose you select a range of text and Delete it, and then Undo that deletion. The editor should not only restore the text, but also restore the selection highlight, so that you can immediately press Delete to delete the same text again.

For consistency, reserve the Undo command for model changes. You can use other commands for view changes. Keep in mind that you don't necessarily need a command named “Undo” to support reversibility. There are other commands that move through other action streams (Back), and physical manipulations (like scrollbar dragging) support direct reversibility.

Users may not even think of reaching for Undo if the rest of your interface makes it easy to reverse undesired changes. Undo is a form of **backward error recovery**, which fixes errors by going back in time. A more natural way of thinking is **forward error recovery** – using other commands to reverse the change. For example, to undo a Bold command by forward error recovery, you select the text again and toggle Bold off. If your interface supports forward error recovery as much as possible, then warts in the Undo model won't hurt as much.

ERROR MESSAGES

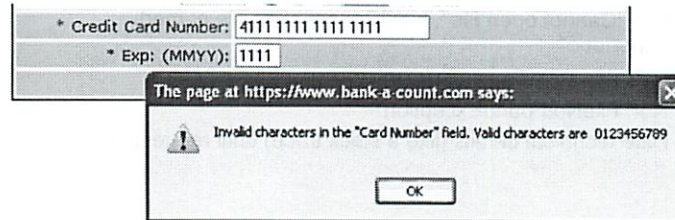
Spring 2012

6.813/6.831 User Interface Design and Implementation

36

Writing Error Message Dialogs

- Best error message is **none at all**
 - Errors should be prevented
 - Be more flexible and tolerant
 - Nonsense entries can often be ignored without harm



Source: "No Dashes Or Spaces" Hall of Shame

Spring 2011

6.813/6.831 User Interface Design and Implementation

37

Finally, let's talk about how to write error messages. But before you try to write an error message, stop and ask yourself whether it's really necessary. An error message is evidence of a limitation or lack of flexibility on the part of the system – a failure to prevent an error or absorb it without complaint. So try to **eliminate the error** first.

Some errors simply aren't worth a message. For example, suppose the user types "abc" into the font size combo box. Don't pop up a message complaining about an "invalid entry". Just ignore it and immediately replace it with the current font size. (Why is this enough feedback, for a font size combo box?) Similarly, if the user drags a scrollbar thumb too far, the scrollbar doesn't pop up an error message ("Too far! Too far!"). It simply stops. If the effect of the erroneous action is easily visible, as in these cases, then you don't have to beat the user over the head with a superfluous error message.

The figure shows an example of an error message that simply shouldn't happen. Forbidding dashes and spaces in a number that the user must type, like an account number or credit card number, is poisonous to usability. (Why are dashes and spaces helpful for human perception and memory?) There's a great collection of error messages like this at the No Dashes or Spaces Hall of Shame (<http://www.unixwiz.net/ndos-shame.html>).

Be Precise and Comprehensible

- Be precise
 - “File missing or wrong format”
 - “File can't be parsed”
 - “Line too long”
 - “Name contains bad characters”
- Restate user's input
 - Not “Cannot open file”, but “Cannot open file named paper.doc”
- Speak the user's language
 - Not “FileNotFoundException”
 - Hide technical details (like a stack trace) until requested

Spring 2011

6.813/6.831 User Interface Design and Implementation

38

Assuming you can't design the error message out of the system, here are some guidelines for writing good ones.

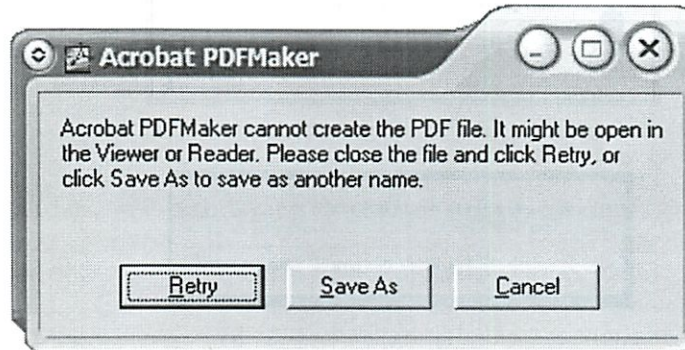
First, **be precise**. Don't lump together multiple error conditions into a single all-purpose message. Find out what's really wrong, and display a targeted message. If the error is due to limitations of your system, like sizes or allowed characters, then be specific about what the limitations are, so that the user can adapt. (Then ask yourself why you have those limitations!)

It often helps to **restate the user's input**, so that they can relate what they did to the error message, and perhaps even detect the problem immediately (“oh, I didn't mean paper.doc...”)

In error messages, it's particularly important to **speak the user's language**, and avoid letting technical terms or details like exceptions and stack traces leak through.

Suggest Reasons and Solutions

- Give **constructive** help
 - why error occurred and how to fix it



Spring 2011

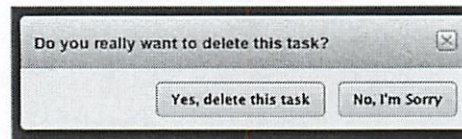
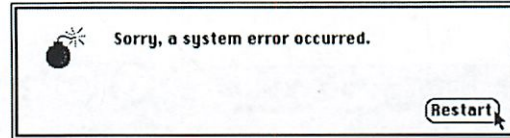
6.813/6.831 User Interface Design and Implementation

39

Next, your message should be **constructive**, not just reporting the error but helping the user correct it. Suggest possible reasons for the error and offer ways to correct them – ideally in the error message dialog itself. Here's a good example from Adobe Acrobat.

Be Polite

- Be polite and nonblaming



Spring 2011

6.813/6.831 User Interface Design and Implementation

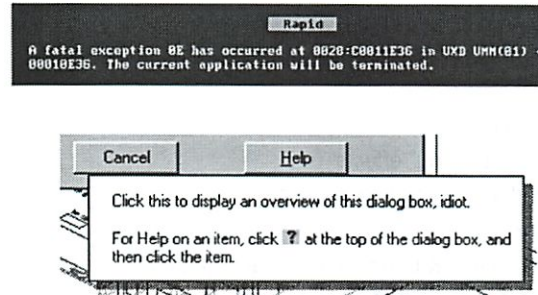
40

Finally, be polite. The message should be worded to take as much blame as possible away from the user and heap the blame instead on the system. Save the user's face; don't worry about the computer's. The computer doesn't feel it, and in many cases it is the interface's fault anyway for not finding a way to prevent the error in the first place. It's interesting to contrast what the original 1984 Mac said when it crashed (an apology!).

The confirmation dialog on the bottom isn't an error message, strictly speaking, but it does show incorrect attribution of blame. The *user* shouldn't have to apologize!

Avoid Loaded Words

- Fatal, illegal, aborted, terminated



Source: Interface Hall of Shame

Spring 2011

6.813/6.831 User Interface Design and Implementation

41

Many words that are unfortunately common in technical error messages have emotionally-charged meanings in ordinary language; examples include “fatal”, “illegal”, “abort”, etc. Avoid them. Use neutral language. Windows and DOS have historically been littered with messages like these.

The tooltip shown at the bottom isn’t strictly an error message, but it actually appeared in a production version of AutoCad! As the story goes, it was inserted by a programmer as a joke, but somehow never removed before release. Even as a joke, it demonstrates a lack of respect for the intelligence of the human being on the other side of the screen. That attitude is exactly wrong for user interface design.

Summary

- Kinds of human error
 - slips & lapses
 - capture, description, mode slips
 - big reasons for error: “strong but wrong”, inattention, efficiency
- Error prevention
 - avoid “strong but wrong” effects
 - confirmation dialogs aren’t a panacea
- Error recovery
 - give the user control & freedom over dialog & data
 - support undo, but know that it can be subtle
- Error messages
 - be specific, be polite, be non-blaming

Victor - team meeting

Not entire poster

- whole poster is a QR code?

Explains ~~that~~ thing

UI design

- minimal on phone

Talk about admin side

(adversarial)

Well beaten path

- some novel

- challenge norms

- creative ways to do ~~new~~ things

- will paper prototype

- try to do something way out there

- forget feasibility

- incorporate interface to UI

6.813 L9
Output

3/2

(2 min late)

Mall of Fame/share Domino Pizzas Builder

* Continuous Visual Representation
Great shortcuts on top

When change size of pizza - size changes

Large preview - tells how many it feeds

- w/ a bit of our lap

But it disappears too fast

Full text based shopping cart

No prices

- for people who care about price

Nano quiz

Description error - features of world confusing
w/ other features

A = laptop leaving debt card
B = capture ing not in
C = pour OJ into milk
D = V picking wrong container
E = laptop walk in room + forget

② Always support (RUD) if possible
Modes

Not undo

Keyboard focus ✓ - which window gets keyboard input

Pen tool ✓ - changes what drag does

Caps lock ✓

Today: Output approaches

Drawing

Rasterization

(one more)

1. View Tree / Objects

2. Strokes

3. Pixels

- everything now

- before (RT gun
or plotter w/ a pen

↗ drawing

↘ rasterization

~~Stack~~
Stack
↓

(3)

At the OS level - window must be an object

You decide where to draw line b/w items

Objects can be pretty heavy

- lots of state + properties
- may not need each of those
- So could draw these as images w/ copy - bit bit

HTML

Primitive objects \rightarrow div + spans

Or use div to instantiate 3rd party ~~all~~ widgets,
- like JQuery slider

CSS

View tree specifies structure

Put presentation into CSS separately

Talk about els w/ CSS selectors

④

Selector language

- tag name button
- ID # main
- class toolbarButton
- element paths # toolbar button

↳ an el. can have multiple classes, sep w/ spaces
(I didn't know this for some time)

Patterns interact in a cascade

- priority order for which stylesheet
- priority order for rules
 - whole page
 - └ - div
 - └ - specific tip
 - └ - inline style
 - JS specified changes

JQuery Dialog

Little div for changing cursor
and event handler so can drag
no purpose for output (blank)

5

CSS image sprite

- lots of imgs in 1

Span w/ text

Span^{text} not printed sometime

↳ text-indent

(Only a few people know these aggressive tricks)

How view tree is drawn

- each section draws itself

- w/ a clip region - object bounding box

- each region is drawn at once

(see slides) when window redraw

- Or only region affected

↳ based on redraw region algorithm

Z-order - how tree is drawn - order

- diff toolkits have diff defaults

- CSS in tree order

- Java Swing in reverse tree order

⑥

Stroke Calls

Usually none in CSS/JS

Is in HTML 5 w/ canvas

Drawing interface to draw this

Antialiasing

Subpixel rendering

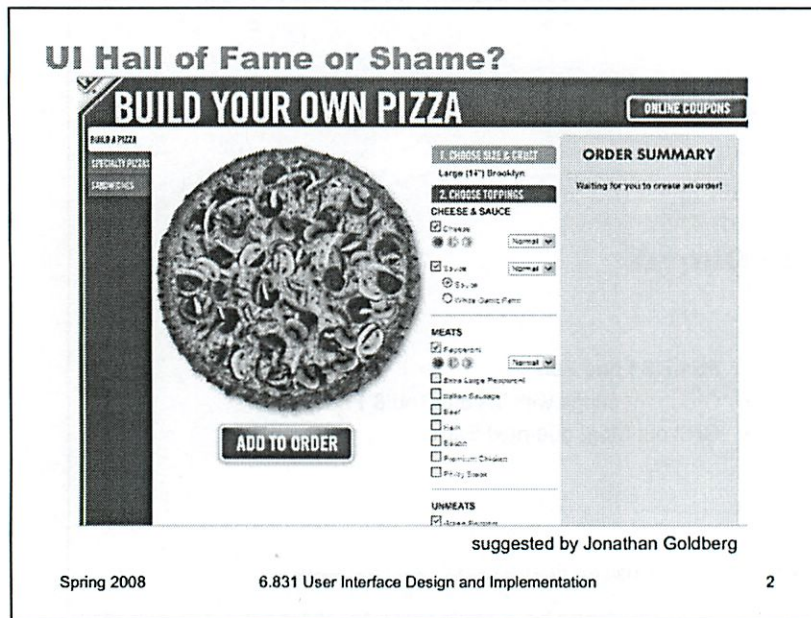
Can see w/ Mac zooming

Gives 3x the horiz resolution

Auto done w/ text

L9: Output

- PS1/RS1 due Sun
- Group meetings with TA this Thu & Fri
- GR2 out Mon, due next Sun



Today's hall of fame or shame candidate is 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).

Some aspects to think about:

- learnability
- visibility
- user control & freedom
- efficiency

Today's Topics

- Output approaches
- Drawing
- Rasterization
- Declarative programming

Spring 2010

6.813/6.831 User Interface Design and Implementation

5

Today's lecture continues our look into the mechanics of implementing user interfaces, by considering **output** in more detail.

One goal for these implementation lectures is not to teach any one particular GUI system or toolkit, but to give a survey of the issues involved in GUI programming and the range of solutions adopted by various systems. Although our examples will generally come from HTML/CSS/Javascript/jQuery, these lectures should give you a sense for what's common and what's unusual in the toolkit you already know, and what you might expect to find when you pick up another GUI toolkit.

Three Output Approaches

- **Objects**
 - Graphical objects arranged in a tree with automatic redraw
 - Example: Label object, Line object
 - Also called: views, interactors, widgets, controls, elements
- **Strokes**
 - High-level drawing primitives: lines, shapes, curves, text
 - Example: drawText() method, drawLine() method
 - Also called: vector graphics, structured graphics
- **Pixels**
 - 2D array of pixels
 - Also called: raster, image, bitmap

Spring 2010

6.813/6.831 User Interface Design and Implementation

6

There are basically three ways to represent the output of a graphical user interface.

Objects is the same as the view tree we discussed previously. Parts of the display are represented by view objects arranged in a spatial hierarchy, with automatic redraw propagating down the hierarchy. There have been many names for this idea over the years; the GUI community hasn't managed to settle on a single preferred term.

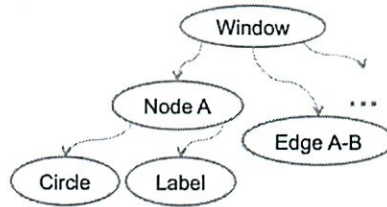
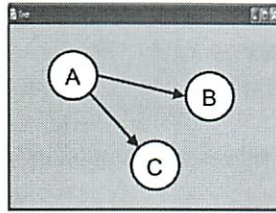
Strokes draws output by making procedure calls to high-level drawing primitives, like drawLine, drawRectangle, drawArc, and drawText.

Pixels regards the screen as an array of pixels and deals with the pixels directly.

All three output approaches appear in virtually every modern GUI application. The object approach always appears at the very top level, for windows, and often for graphical objects within the windows as well. At some point, we reach the leaves of the view hierarchy, and the leaf views draw themselves with stroke calls. A graphics package then converts those strokes into pixels displayed on the screen. For performance reasons, an object may short-circuit the stroke package and draw pixels on the screen directly. On Windows, for example, video players do this using the DirectX interface to have direct control over a particular screen rectangle.

What approach do each of the following representations use? HTML (object); Postscript laser printer (stroke input, pixel output); plotter (stroke input and output); PDF (stroke); LCD panel (pixel).

Example: Designing a Graph View



- Object approach
 - Each node and edge is an object in the view tree
 - A node object might have two child objects: circle and label
- Stroke approach
 - Graph view draws lines, circles and text
- Pixel approach
 - Graph view has pixel images of the nodes

Spring 2010

6.813/6.831 User Interface Design and Implementation

7

Since virtually every GUI uses all three approaches, the design question becomes: at which points in your application do you want to step down into a lower-level kind of output? Here's an example. Suppose you want to build a view that displays a graph of nodes and edges.

One way to do it would represent each node and each edge in the graph by a component (as in the tree on the right). Each node in turn might have two components, a circle and a text label. Eventually, you'll get down to the primitive objects available in your GUI toolkit. Most GUI toolkits provide a text label; most don't provide a primitive circle. (One notable exception is SVG, which has component equivalents for all the common drawing primitives.) This would be a **pure object approach**, at least from your application's point of view – stroke output and pixel output would still happen, but inside primitive objects that you took from the library.

Alternatively, the top-level window might have *no* subcomponents. Instead, it would draw the entire graph by a sequence of stroke calls: `drawCircle` for the node outlines, `drawText` for the labels, `drawLine` for the edges. This would be a **pure stroke**.

Finally, your graph view might bypass stroke drawing and set pixels in the window directly. The text labels might be assembled by copying character images to the screen. This **pure pixel approach** is rarely used nowadays, because it's the most work for the programmer, but it used to be the only way to program graphics.

Hybrid approaches for the graph view are certainly possible, in which some parts of the output use one approach, and others use another approach. The graph view might use objects for nodes, but draw the edges itself as strokes. It might draw all the lines itself, but use label objects for the text.

Issues in Choosing Output Approaches

- Layout
- Input
- Redraw
- Drawing order
- Heavyweight objects
- Device dependence

Spring 2010

6.813/6.831 User Interface Design and Implementation

8

Layout: Objects remember where they were put, and draw themselves there. They also support automatic layout. With strokes or pixels, you have to figure out (at drawing time) where each piece goes, and put it there.

Input: Objects participate in event dispatch and propagation, and the system automatically does **hit-testing** (determining whether the mouse is over the component when an event occurs) for objects, but not for strokes. If a graph node is an object, then it can receive its own click and drag events. If you stroked the node instead, then you have to write code to determine which node was clicked or dragged.

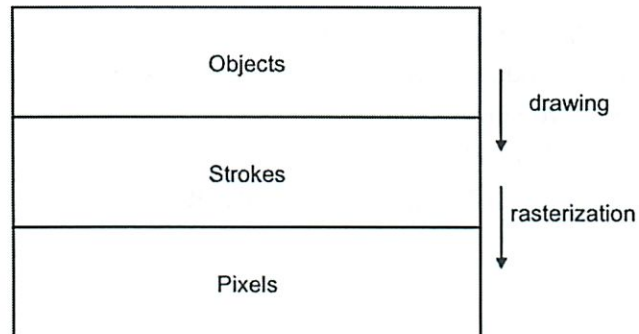
Redraw: An automatic redraw algorithm means that components redraw themselves automatically when they have to. Furthermore, the redraw algorithm is efficient: it only redraws components whose extents intersect the damaged region. The stroke or pixel model would have to do this test by hand. In practice, most stroked objects don't bother, simply redrawing everything whenever some part of the view needs to be redrawn.

Drawing order: It's easy for a parent to draw before (underneath) or after (on top of) all of its children. But it's not easy to interleave parent drawing with child drawing. So if you're using a hybrid model, with some parts of your view represented as components and others as strokes, then the components and strokes generally fall in two separate layers, and you can't have any complicated layering relationships between strokes and components.

Heavyweight objects: Objects may be big -- even an object with no fields costs about 20 bytes in Java. As we've seen, the view tree is overloaded not just with drawing functions but also with event dispatch, automatic redraw, and automatic layout, so the properties and state used by those processes further bulks up the class. Views derived from large amounts of data -- say, a 100,000-node graph -- generally can't use an object for every individual data item. The "flyweight" pattern can help, by storing redundant information in the object's context (i.e., its parent) rather than in each component, but few toolkits support flyweight objects. (See [Glyphs: Flyweight Objects for User Interfaces](#) by Paul R. Calder and Mark A. Linton. *UIST '90*.)

Device dependence: The stroke approach is largely device independent. In fact, it's useful not just for displaying to screens, but also to printers, which have dramatically different resolution. The pixel approach, on the other hand, is extremely device dependent. A directly-mapped pixel image won't look the same on a screen with a different resolution.

How Output Approaches Interact



Spring 2010

6.813/6.831 User Interface Design and Implementation

9

As we said earlier, almost every GUI program uses all three approaches. At the highest level, a typical program presents itself in a window, which is an object. At the lowest level, the window appears on the screen as a rectangle of pixels. So a series of steps has to occur that translates that window object (and all its descendents in the view tree) into pixels.

The step from objects down to strokes is usually called **drawing**. We'll look at that first.

The step from strokes down to pixels is called **rasterization** (or scan conversion). The specific algorithms that rasterize various shapes are beyond the scope of this course (see 6.837 Computer Graphics instead). But we'll talk about some of the effects of rasterization, and what you need to know as a UI programmer to control those effects.

Object Approach in HTML

- Instantiate a view element
 - built-in widget `<button>`, `<input type="text">`
 - primitive `<div>`, ``
 - third-party widget `<div class="ui-slider">`
- Set its output behavior using styles

Cascading Style Sheets (CSS)

- Key idea: separate the **structure** of the UI (view tree) from details of **presentation**
 - HTML is structure, CSS is presentation
- Two ways to use CSS
 - As an attribute of a particular HTML element
`<button style="font-weight:bold;"> Cut </button>`
 - As a style sheet defining style rules for many HTML elements at once
`<style>
 button { font-weight:bold; }
</style>`

Spring 2011

6.813/6.831 User Interface Design and Implementation

11

Our second example of declarative specification is Cascading Style Sheets, or CSS. Where HTML creates a view hierarchy, CSS adds style information to the hierarchy – fonts, colors, spacing, and layout.

There are two ways to use CSS. The first way is by setting styles directly on individual objects. The style attribute of any HTML element can contain a set of CSS settings (which are simply **name:value** pairs separated by semicolons).

The second way is more interesting, because it's more declarative. Rather than finding each individual component and directly setting its style attribute, you specify a **style sheet** that defines rules for assigning styles to elements. Each rule consists of a pattern that matches a set of HTML elements, and a set of CSS definitions that specify the style for those elements. In this simple example, **button** matches all the button elements, and the body of the rule sets them to boldface font.

The style sheet is included in the HTML by a `<style>` element, which either embeds the style sheet as text between `<style>` and `</style>`, or refers to a URL that contains the actual style sheet.

CSS Selectors

- Each rule in a style sheet has a **selector** pattern that matches a set of HTML elements

Tag name

button { font-weight:bold; }

ID

#main { background-color:
rgb(100%,100%,100%); }

Class attribute

.toolbarButton { font-size: 12pt; }

Element paths

#toolbar button { display: hidden; }

```
<div id="main">  
  <div id="toolbar">  
    <button class="toolbarButton">  
      </img>  
    </button>  
  </div>  
  <textarea id="editor"></textarea>  
</div>
```

The pattern in a CSS rule is called a **selector**. The language of selectors is simple but powerful. Here are a couple of the more common selectors. Selectors are also used by jQuery to select and operate on nodes in the DOM tree, so it's worth becoming familiar with this pattern language.

Cascading and Inheritance

- If multiple rules apply to the same element, rules are automatically combined with **cascading precedence**
 - Source: browser defaults < web page < user overrides
Browser says: a { text-decoration: underline; }
Web page says: a { text-decoration: none; }
User says: a { text-decoration: underline; }
 - Rule specificity: general selectors < specific selectors
 button { font-size: 12pt; }
 .toolbarButton { font-size: 14pt; }
- Styles can also be **inherited** from element's parent
 - This is the default for simple styles like font, color, and text properties
 body { font-size: 12pt; }

Spring 2011

6.813/6.831 User Interface Design and Implementation

13

There can be multiple style sheets affecting an HTML page, and multiple rules within a style sheet. Each rule affects a set of HTML elements, so what happens when an element is affected by more than one rule? If the rules specify independent style properties (e.g., one rule specifies font size, and another specifies color), then the answer is simple: both rules apply. But what if the rules *conflict* with each other – e.g., one says the element should be bold, and another says it shouldn't?

To handle these cases, declarative rule-based systems need a conflict resolution mechanism, and CSS is no different. CSS's resolution mechanism is called **cascading** (hence the name, Cascading Style Sheets). It has two main resolution strategies. The overall idea is that more specific rules should take precedence over more general rules. This is reflected first in where the style sheet rule came from: some rules are web browser defaults, for all users and all web pages; others are defaults set by a specific user for all web pages; others are provided by a specific web page in a <style> element. In general, the web page rule wins (although the user can override this by setting the priority of their own CSS rules to *important*). Second, rules with more specific selectors (like specific element IDs or class names) take precedence over rules with more general selectors (like element names).

This is an example of why declarative specification is powerful. A single rule – like a user override – can affect a large swath of the behavior of the system, without having to write a lot of procedural code, and without having to make sure that procedural code runs at just the right time.

But it also illustrates the difficulties of debugging declarative specifications. You may add a rule to the style sheet, maybe trying to change a button's font size, only to see *no change* in the result – because some other rule that you aren't aware of is taking precedence. CSS conflict resolution is a complex process that may require trial-and-error to debug.

Declarative Styles vs. Procedural Styles

CSS

```
// found in a <style> element  
button { font-size: 12pt; font-weight: bold; }
```

jQuery

```
// in a <script> element  
$("button").css("font-size", "12pt").css("font-weight", "bold");
```

Just as with HTML, we can change CSS styles procedurally as well. jQuery offers a particularly nice way to do this, which matches very closely the parts of a CSS rule: a selector, a property name, and a value.

Exercise: Explore How Widgets Are Drawn

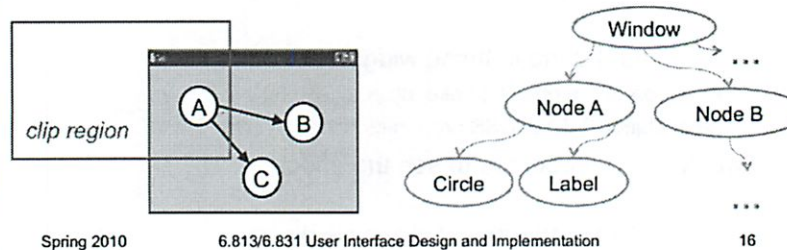
- Go to jqueryui.com/demos
- Open your browser's developer console
- Examine these widgets:



- How is output done in these widget?
 - Which parts are object, stroke, pixel?
 - What objects? What CSS properties?
- Tweak some properties to see the effect

How a View Tree is Drawn

- Drawing goes top down
 - Draw self (using strokes or pixels)
 - For each child component,
 - If child intersects clipping region then
 - intersect clipping region with child's bounding box
 - recursively draw child with clip region set to the intersection



Spring 2010

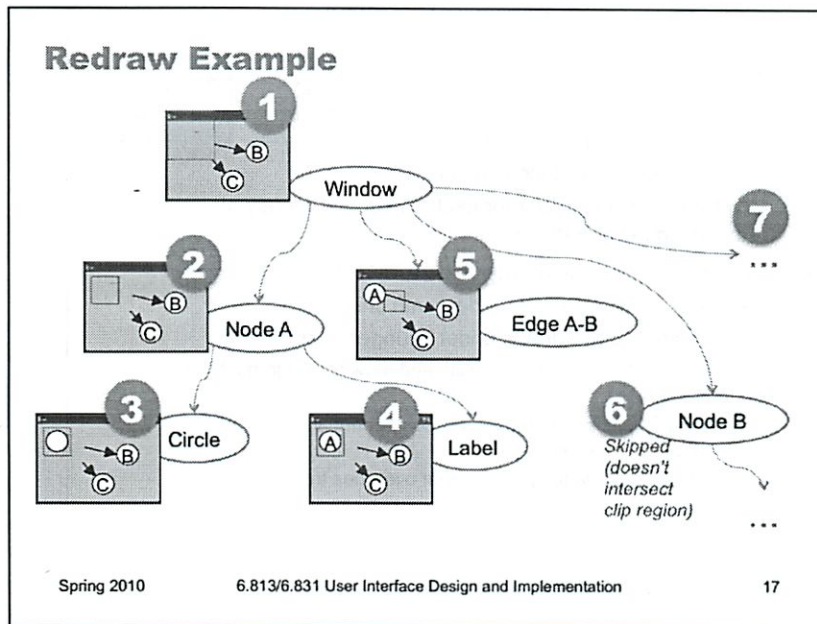
6.813/6.831 User Interface Design and Implementation

16

Here's how drawing works in the object approach. Drawing is a top-down process: starting from the root of the view tree, each object draws itself, then draws each of its children recursively. The process is optimized by passing a **clipping region** to each object, indicating the area of the screen that needs to be drawn. Children that do not intersect the clipping region are simply skipped, not drawn. In the example above, nodes B and C would not need to be drawn. When an object partially intersects the clipping region, it must be drawn – but any strokes or pixels it draws when the clipping region is in effect will be masked against the clip region, so that only pixels falling inside the region actually make it onto the screen.

For the root, the clipping region might be the entire screen. As drawing descends the tree, however, the clipping region is intersected with each object's bounding box. So the clipping region for an object deep in the tree is the intersection of the bounding boxes of its ancestors.

For high performance, the clipping region is normally rectangular, using **bounding boxes** rather than the graphical object's actual shape. But it doesn't have to be that way. A clipping region can be an arbitrary shape on the screen. This can be very useful for visual effects: e.g., setting a string of text as your clipping region, and then painting an image through it like a stencil. Postscript was the first stroke model to allow this kind of nonrectangular clip region. Now many graphics toolkits support nonrectangular clip regions. For example, on Microsoft Windows and X Windows, you can create nonrectangular windows, which clip their children into a nonrectangular region.



Here's an example of the redraw algorithm running on the graph window (starting with the clipping region shown on the last slide).

1. First the clip region is intersected with the whole window's bounding box, and the window is told to draw itself within that intersection. The window draws its titlebar and its gray background. The window background effectively erases the previous contents of the window.

2. The window's clip region is now intersected with its first child's bounding box (Node A), and Node A is told to draw itself within that. In this particular example (where nodes are represented by circle and label components), Node A doesn't do any of its own drawing; all the drawing will be handled by its children.

3. Now Node A's circle child is told to draw itself. In this case, the circle has the same bounding box as Node A itself, so it receives the same clip region that Node A did. It draws a white circle.

4. Now Node A's label child is told to draw itself, again using the same clip region because it has the same bounding box. It draws text on top of the circle just drawn.

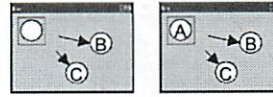
5. Popping back up the tree, the next child of the window, Edge A-B, is told to draw itself, using the clip region that intersects its own bounding box with the window's clip region. Only part of the edge falls in this clip region, so the edge only draws part of itself.

6. The next child of the window, Node B, doesn't intersect the window's clip region at all, so it isn't told to draw itself.

7. The algorithm continues through the rest of the tree, either drawing children or skipping them depending on whether they intersect the clip region. (Would Edge A-C be drawn? Would Node C be drawn?)

Note that the initial clip region passed to the redraw algorithm will be different every time the algorithm is invoked. Clip regions generally come from *damage rectangles*, which will be explained in a moment.

Z Order



- 2D GUIs are really “2 ½ D”
 - Drawing order produces layers
 - Not a true z coordinate for each object, but merely an **ordering** in the z dimension
- View tree and redraw algorithm dictate z order
 - Parents are drawn first, underneath children
 - Older siblings are drawn under younger ones
 - Flex, HTML, most GUI toolkits and drawing programs behave this way
 - Java Swing is backwards: last component added (highest index) is drawn first
 - CSS has a z-index property that overrides tree structure

Spring 2010

6.813/6.831 User Interface Design and Implementation

18

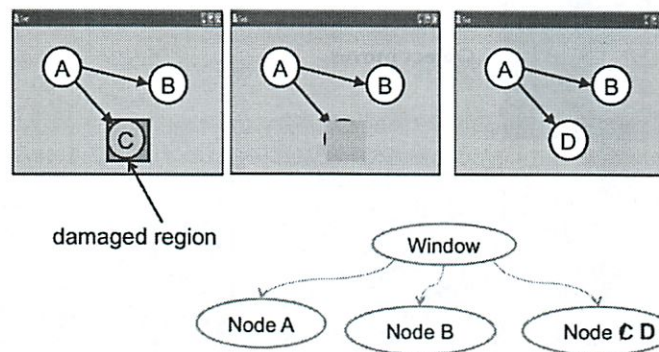
When the bounding boxes of two objects overlap, like the circle and label components in the previous example, the redraw algorithm induces an ordering on the objects that makes them appear layered, one on top of the other. For this reason, 2D graphical user interfaces are sometimes called 2½D. They aren’t fully 3D, in which objects have x, y, and z coordinates; instead the z dimension is merely an ordering, called **z order**.

Z order is a side-effect of the order that the objects are drawn when the redraw algorithm passes over the tree. Since drawing happens top-down, parents are generally drawn underneath children (although parents get control back after their children finish drawing, so a parent can draw some more on top of all its children if it wants). Older siblings (with lower indexes in their parent’s array of children) are generally drawn underneath younger ones. Java Swing is a curious exception to this – its redraw algorithm draws the highest-index child first, so the youngest sibling ends up on the bottom of the z order.

Z order can be affected by rearranging the tree, e.g. moving children to a different index position within their parent, or promoting them up the tree if necessary. This is often important for operations like drag-and-drop, since we generally want the object being dragged to appear on top of other objects.

Some GUI toolkits allow you to change the z-order of an element without moving its position in the tree. In HTML, the CSS z-index property lets you do that. There’s a nice page (http://tjkdesign.com/articles/z-index/teach_yourself_how_elements_stack.asp) that lets you explore how the z-index property works.

Damage and Automatic Redraw



Spring 2010

6.813/6.831 User Interface Design and Implementation

19

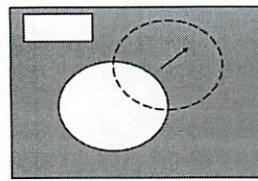
When a graphical object needs to change its appearance, it doesn't repaint itself directly. It *can't*, because the drawing process has to occur top-down through the view tree: the object's ancestors and older siblings need to have a chance to paint themselves underneath it. (So, in Java, even though a graphical object can call its own `paint()` method directly, you generally shouldn't do it!)

Instead, the object asks the graphics system to repaint it at some time in the future. This request includes a **damaged region**, which is the part of the screen that needs to be repainted. Often, this is just the entire bounding box of the object; but complex objects might figure out which part of the screen corresponds to the part of the model that changed, so that only that part is damaged.

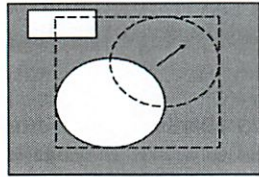
The repaint request is then **queued** for later. Multiple pending repaint requests from different objects are consolidated into a single damaged region, which is often represented just as a rectangle – the bounding box of all the damaged regions requested by individual objects. That means that undamaged screen area is being considered damaged, but there's a tradeoff between the complexity of the damaged region representation and the cost of repainting.

Eventually – usually after the system has handled all the input events (mouse and keyboard) waiting on the queue -- the repaint request is finally satisfied, by setting the clipping region to the damaged region and redrawing the view tree from the root.

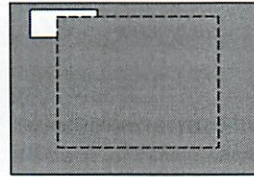
Naïve Redraw Causes Flashing Effects



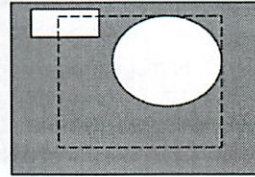
Object moves



**Determine
damaged region**



**Redraw parent
(children blink out!)**



Redraw children

Spring 2010

6.813/6.831 User Interface Design and Implementation

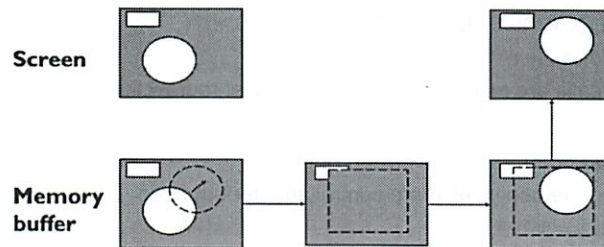
20

There's an unfortunate side-effect of the automatic damage/redraw algorithm. If we draw a view tree directly to the screen, then moving an object can make the screen appear to flash – objects flickering while they move, and nearby objects flickering as well.

When an object moves, it needs to be erased from its original position and drawn in its new position. The erasure is done by redrawing all the objects in the view hierarchy that intersect this damaged region; typically the drawing of the window background is what does the actual erasure. If the drawing is done directly on the screen, this means that all the objects in the damaged region temporarily *disappear*, before being redrawn. Depending on how screen refreshes are timed with respect to the drawing, and how long it takes to draw a complicated object or multiple layers of the hierarchy, these partial redraws may be briefly visible on the monitor, causing a perceptible flicker.

Double-Buffering

- Double-buffering solves the flashing problem



Spring 2010

6.813/6.831 User Interface Design and Implementation

21

Double-buffering solves this flickering problem. An identical copy of the screen contents is kept in a memory buffer. (In practice, this may be only the part of the screen belonging to some subtree of the view hierarchy that cares about double-buffering.) This memory buffer is used as the drawing surface for the automatic damage/redraw algorithm. After drawing is complete, the damaged region is just copied to screen as a block of pixels. Double-buffering reduces flickering for two reasons: first, because the pixel copy is generally faster than redrawing the view hierarchy, so there's less chance that a screen refresh will catch it half-done; and second, because unmoving objects that happen to be caught, as innocent victims, in the damaged region are never erased from the screen, only from the memory buffer.

It's a waste for every individual view to double-buffer itself. If any of your ancestors is double-buffered, then you'll derive the benefit of it. So double-buffering is usually applied to top-level windows.

Why is it called double-buffering? Because it used to be implemented by two interchangeable buffers in video memory. While one buffer was showing, you'd draw the next frame of animation into the other buffer. Then you'd just tell the video hardware to switch which buffer it was showing, a very fast operation that required no copying and was done during the CRT's vertical refresh interval so it produced no flicker at all.

Going From Objects to Strokes

- Drawing method approach
 - e.g. Swing `paint()` method
 - Drawing method is called directly during redraw; override it to change how component draws itself
- Retained graphics approach
 - e.g. Adobe Flex
 - Stroke calls are recorded and played back at redraw time
- Differences
 - Retained graphics is less error prone
 - Drawing method gives more control and performance

Spring 2010

6.813/6.831 User Interface Design and Implementation

22

In our description of the redraw algorithm, we said a graphical object “draws itself,” meaning that it produces strokes to show itself on the screen. How that is actually done depends on the GUI toolkit you’re using.

In Java Swing (and many other desktop GUI toolkits, like Win32 and Cocoa), every object has a **drawing method**. In Swing, this method is `paint()`. The redraw algorithm operates by recursively calling `paint()` down the view hierarchy. Objects can override the `paint()` method to change how they draw themselves. In fact, Swing breaks the `paint()` method down into several overridable template methods, like `paintComponent()` and `paintChildren()`, to make it easier to affect different parts of the redraw process. More about Swing’s painting process can be found in “Painting in AWT and Swing” by Amy Fowler (<http://java.sun.com/products/jfc/tsc/articles/painting/>).

In Adobe Flex, there’s no drawing method available to override – the redraw algorithm is *hidden* from the programmer, much like the event loop is hidden by these toolkits. Instead, you make a sequence of stroke calls into the object, and the object records this sequence of calls. Subsequently, whenever the object needs to redraw itself, it just plays back the recorded sequence of stroke calls. This approach is sometimes called **retained graphics**.

A key difference between these approaches is **when** stroke calls can be made. With the drawing method approach, drawing should *only* be done while the drawing method is active. Drawing done at a different time (like during an event handler) will not interact correctly with the redraw algorithm; it won’t respect z order, and it will be ephemeral, overwritten and destroyed the next time the redraw algorithm touches that object. With the retained graphics approach, however, the stroke calls can be recorded at any time, and the toolkit automatically handles playing them back at the right point in the redraw. The retained graphics approach tends to be less error prone for a programmer; drawing at the wrong time is a common mistake for beginning Swing programmers.

A potential downside of the retained graphics approach is performance. The recorded strokes must be stored in memory. Although this recording is not as heavyweight as a view tree (since it doesn’t have to handle input or layout, or even necessarily be represented as objects), you probably wouldn’t want to do it with millions of stroke calls. So if you had an enormous view (like a map) being displayed inside a scrolling pane (so that only a small part of it was visible on screen), you wouldn’t want to stroke the entire map. The drawing method approach gives more control over this; since you have access to the clip region in the drawing method, you can choose not to render strokes that would be clipped. To do the equivalent thing with retained graphics would put more burden on the programmer to determine the visible rectangle and rerecord the stroke calls every time this rectangle changed.

Stroke Model

- Drawing surface
 - Also called drawable (X Windows), GDI (MS Win)
 - Screen, memory buffer, print driver, file, remote screen
- Graphics context
 - Encapsulates drawing parameters so they don't have to be passed with each call to a drawing primitive
 - Font, color, line width, fill pattern, etc.
- Coordinate system
 - Origin, scale, rotation
- Clipping region
- Drawing primitives
 - Line, circle, ellipse, arc, rectangle, text, polyline, shapes

Spring 2010

6.813/6.831 User Interface Design and Implementation

23

Now let's look at the drawing capabilities provided by the stroke model.

Every toolkit's stroke model has some notion of a **drawing surface**. The screen is only one possible place where drawing might go. Another common drawing surface is a memory buffer, which is an array of pixels just like the screen. Unlike the screen, however, a memory buffer can have arbitrary dimensions. The ability to draw to a memory buffer is essential for double-buffering. Another target is a printer driver, which forwards the drawing instructions on to a printer. Although most printers have a pixel model internally (when the ink actually hits the paper), the driver often uses a stroke model to communicate with the printer, for compact transmission. Postscript, for example, is a stroke model.

Most stroke models also include some kind of a **graphics context**, an object that bundles up drawing parameters like color, line properties (width, end cap, join style), fill properties (pattern), and font.

The stroke model may also provide a current **coordinate system**, which can be translated, scaled, and rotated around the drawing surface. We've already discussed the **clipping region**, which acts like a stencil for the drawing. Finally, a stroke model must provide a set of **drawing primitives**, function calls that actually produce graphical output.

Many systems combine all these responsibilities into a single object. Java's Graphics object is a good example of this approach. In other toolkits, the drawing surface and graphics context are independent objects that are passed along with drawing calls.

When state like graphics context, coordinate system, and clipping region are embedded in the drawing surface, the surface must provide some way to save and restore the context. A key reason for this is so that parent views can pass the drawing surface down to a child's draw method without fear that the child will change the graphics context. In Java, for example, the context can be saved by Graphics.create(), which makes a copy of the Graphics object. Notice that this only duplicates the graphics context; it doesn't duplicate the drawing surface, which is still the same.

HTML5 Canvas in One Slide

HTML element

```
<canvas width=1000  
  height=1000></canvas>
```

graphics context

```
var ctx = canvas.getContext  
  ("2d")
```

coordinate system

```
ctx.translate()  
ctx.rotate()  
ctx.scale()
```

color, font, line style, etc.

```
ctx.strokeStyle = "rgb(50%,50%,  
  50%)"  
ctx.fillStyle = ...  
ctx.font = "bold 12pt sans-  
  serif"  
ctx.lineWidth = 2.5
```

drawing primitives

```
ctx.beginPath();  
ctx.moveTo(0,0)  
ctx.lineTo(500,500)  
ctx.stroke()
```

```
ctx.beginPath()  
ctx.arc(500,500,100,0,  
  2*Math.PI,false)  
ctx.fill()
```

clipping

```
ctx.beginPath()  
ctx.rect(0, 0, 200, 300)  
ctx.clip()
```

Spring 2011

6.813/6.831 User Interface Design and Implementation

24

As an example of a stroke library, HTML5 has an element called `<canvas>` that provides a stroke drawing context for Javascript programs. The way to think about a canvas is as a pixel image that you're drawing stroke calls on. The canvas element takes width and height attributes that specify the size of this pixel image. When the canvas is laid out on screen, however, it might be given a different width and height by CSS layout – in which case it will be stretched or shrunk appropriately.

The canvas element provides a graphics context object that you can interact with from Javascript. (The "2d" graphics context is shown here; future canvas implementations may offer 3D rendering contexts.) The interface for this object has all the pieces of a stroke library that we talked about on the previous slide.

Exercise: Try Some Canvas

- Go to htmledit.squarefree.com
- Make a canvas and get its graphics context:

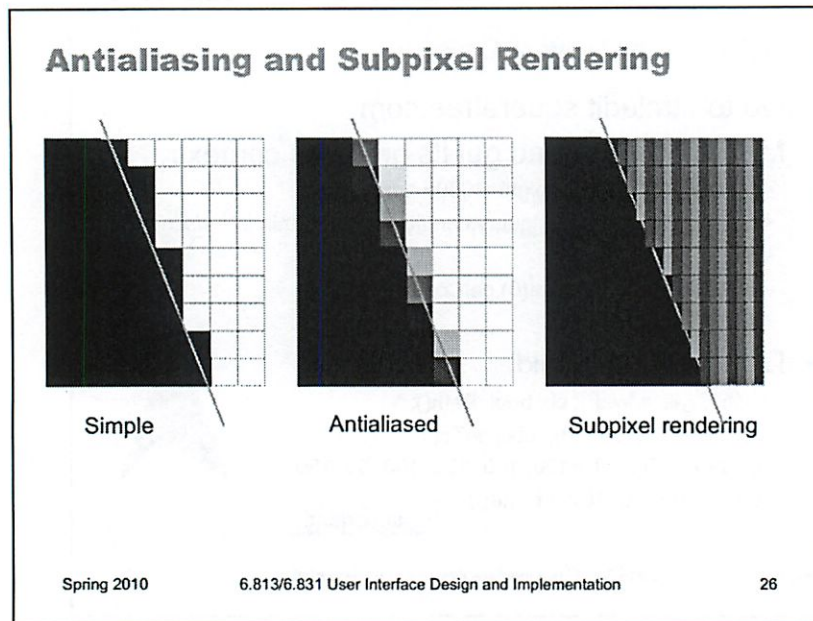
```
<canvas width=1000 height=1000></canvas>  
<script src="http://code.jquery.com/jquery-1.7.1.min.js"></script>  
<script>  
var ctx = $("canvas").get(0).getContext("2d")  
</script>
```

- Draw a delta shield

```
ctx.fillStyle = "red"; ctx.beginPath();  
ctx.moveTo(100, 10); ctx.lineTo(150, 150);  
ctx.bezierCurveTo(100, 100, 130, 100, 50, 150);  
ctx.lineTo(100, 10); ctx.fill();
```

important!





It's beyond the scope of this lecture to talk about algorithms for converting a stroke into pixels. But you should be aware of some important techniques for making strokes look good.

One of these techniques is **antialiasing**, which is a way to make an edge look smoother. Instead of making a binary decision between whether to color a pixel black or white, antialiasing uses a shade of gray whose value varies depending on how much of the pixel is covered by the edge. In practice, the edge is between two arbitrary colors, not just black and white, so antialiasing chooses a point on the gradient between those two colors. The overall effect is a fuzzier but smoother edge.

Subpixel rendering takes this a step further. Every pixel on an LCD screen consists of three discrete pixels side-by-side: red, green, and blue. So we can get a horizontal resolution which is three times the nominal pixel resolution of the screen, simply by choosing the colors of the pixels along the edge so that the appropriate subpixels are light or dark. It only works on LCD screens, not CRTs, because CRT pixels are often arranged in triangles, and because CRTs are analog, so the blue in a single "pixel" usually consists of a bunch of blue phosphor dots interspersed with green and red phosphor dots. You also have to be careful to smooth out the edge to avoid color fringing effects on perfectly vertical edges. And it works best for high-contrast edges, like this edge between black and white. Subpixel rendering is ideal for text rendering, since text is usually small, high-contrast, and benefits the most from a boost in horizontal resolution. Windows XP includes ClearType, an implementation of subpixel rendering for Windows fonts. (For more about subpixel rendering, see Steve Gibson, "Sub-Pixel Font Rendering Technology", <http://grc.com/cleartype.htm>)

Exercise (requires a Mac)

- Turn on screen zoom
 - System Preferences / Mouse
- Turn off smoothing
 - System Preferences / Mouse / Screen Zoom / Options
- Find some text and zoom in on it
- What do you see?

☒ Screen Zoom Options...

☐ Smooth images (Press $\text{⌘} \text{⌘}$ to turn smoothing on or off)



Pixel Approach

- Pixel approach is a rectangular array of pixels
 - Each pixel is a vector (e.g., red, green, blue components), so pixel array is really 3 dimensional
- Bits per pixel (bpp)
 - 1 bpp: black/white, or bit mask
 - 4-8 bpp: each pixel is an index into a color palette
 - 24 bpp: 8 bits for each color
 - 32 bpp: 8 bits for each color + alpha channel
- Color components (e.g. RGB) are also called channels or bands
- Pixel model can be arranged in many ways
 - Packed into words (RRGB GBRG ...) or loosely (RGB- RGB- ...)
 - Separate planes (RRR...GGG...BBB...) vs. interleaved (RGB RGB RGB...)
 - Scanned from top to bottom vs. bottom to top

Spring 2010

6.813/6.831 User Interface Design and Implementation

28

Finally, let's talk in more detail about what a pixel image looks like.

Put simply, it's a rectangular array of pixels – but pixels themselves are not always so simple. A pixel itself has a **depth**, so this model is really three dimensional. Depth is often expressed in **bits per pixel (bpp)**. The simplest kind of pixel model has 1 bit per pixel; this is suitable for representing black and white images. It's also used for **bitmasks**, where the single-bit pixels are interpreted as boolean values (pixel present or pixel missing). Bitmasks are useful for clipping – you can think of a bitmask as a stencil.

Another kind of pixel representation uses each pixel value as an index into a palette, which is just a list of colors. In the 4-bpp model, for example, each of the 16 possible pixel values represents a different color. This kind of representation, often called Indexed Color, was useful when memory was scarce; you still see it in the GIF file format, but otherwise it isn't used much today.

The most common pixel representation is often called “true color” or “direct color”; in this model, each pixel represents a color directly. The color value is usually split up into multiple components: red, green, and blue. (Color components are also called **channels** or **bands**; the red channel of an image, for example, is a rectangular array of the red values of its pixels.)

A pixel model can be arranged in memory (or a file) in various ways: packed tightly together to save memory, or spread out loosely for faster access; with color components interleaved or separated; and scanned from the top (so that the top-left pixel appears first) or the bottom (the bottom-left pixel appearing first).

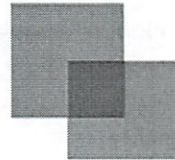
Transparency

- **Alpha** is a pixel's transparency
 - from 0.0 (transparent) to 1.0 (opaque)
 - so each pixel has red, green, blue, and alpha values
- Uses for alpha
 - Antialiasing
 - Nonrectangular images
 - Translucent components
 - Clipping regions with antialiased edges

Many pixel models have a fourth channel in addition to red, green, and blue: the pixel's **alpha** value, which represents its degree of transparency. We'll talk more about alpha in a future lecture.

Exercise: Translucent Rectangles

- Go to htmledit.squarefree.com
- Make a translucent rectangle with `<div>`
`<div style="width:100px; height:100px; background-color:red; opacity:0.5; position:absolute"></div>`
- Make an overlapping rectangle with `<canvas>`
`<canvas width=1000 height=1000 style="position:absolute"></canvas>`
`<script src="http://code.jquery.com/jquery-1.7.1.min.js"></script>`
`<script>`
`var ctx = $("#canvas").get(0).getContext("2d")`
`ctx.fillStyle = "rgba(100%,0%,0%,0.5)"`
`ctx.rect(50,50,100,100)`
`ctx.fill()`
`</script>`



BitBlt

- BitBlt (bit block transfer) copies a block of pixels from one image to another
 - Drawing images on screen
 - Double-buffering
 - Scrolling
 - Clipping with nonrectangular masks
- Compositing rules control how pixels from source and destination are combined
 - More about this in a later lecture

Spring 2010

6.813/6.831 User Interface Design and Implementation

31

The primary operation in the pixel model is copying a block of pixels from one place to another – often called **bitblt** (pronounced “bit blit”). This is used for drawing pictures and icons on the screen, for example. It’s also used for double-buffering – after the offscreen buffer is updated, its contents are transferred to the screen by a bitblt.

Bitblt is also used for screen-to-screen transfers. To do fast scrolling, for example, you can bitblt the part of the window that doesn’t change upwards or downwards, to save the cost of redrawing it. (For example, look at Swing’s `JViewport.BLIT_SCROLL_MODE`.)

It’s also used for sophisticated drawing effects. You can use bitblt to combine two images together, or to combine an image with a mask, in order to clip it or composite them together.

Bitblt isn’t always just a simple array copy operation that replaces destination pixels with source pixels. There are various different rules for combining the destination pixels with the source pixels. These rules are called **compositing** (**alpha compositing**, when the images have an alpha channel), and we’ll talk about them in a later lecture.

Image File Formats

- GIF
 - 8 bpp, palette uses 24-bit colors
 - 1 color in the palette can be transparent (1-bit alpha channel)
 - lossless compression
 - suitable for screenshots, stroked graphics, icons
- JPEG
 - 24 bpp, no alpha
 - lossy compression: visible artifacts (dusty noise, moire patterns)
 - suitable for photographs
- PNG
 - lossless compression
 - 1, 2, 4, 8 bpp with palette
 - 24 or 48 bpp with true color
 - 32 or 64 bpp with true color and alpha channel
 - suitability same as GIF
 - better than GIF, but no animation

Spring 2010

6.813/6.831 User Interface Design and Implementation

32

Here are a few common image file formats. It's important to understand when to use each format. For user interface graphics, like icons, JPG generally should *not* be used, because it's lossy compression – it doesn't reproduce the original image exactly. When every pixel matters, as it does in an icon, you don't want lossy compression. JPG also can't represent transparent pixels, so a JPG image always appears rectangular in your interface.

For different reasons, GIF is increasingly unsuitable for interface graphics. GIF isn't lossy – you get the same image back from the GIF file that you put into it – but its color space is very limited. GIF images use 8-bit color, which means that there can be at most 256 different colors in the image. That's fine for some low-color icons, but not for graphics with gradients or blurs. GIF has limited support for transparency – pixels can either be opaque (alpha 1) or transparent (alpha 0), but not *translucent* (alpha between 0 and 1). So you can't have fuzzy edges in a GIF file, that blend smoothly into the background. GIF files can also represent simple animations.

PNG is the best current format for interface graphics. It supports a variety of color depths, and can have a full alpha channel for transparency and translucency. (Unfortunately Internet Explorer 6 doesn't correctly display transparent PNG images, so GIF still rules web graphics.)

If you want to take a screenshot, PNG is the best format to store it.

Hints for Debugging Output

- Something you're drawing isn't appearing on the screen. Why not?
 - Wrong visibility setting
 - CSS display property
 - Wrong place
 - left/top, position properties
 - Wrong size
 - width/height
 - Wrong color
 - color, background-color, background-image
 - Wrong z-order

Spring 2010

6.813/6.831 User Interface Design and Implementation

33

A final word about debugging the output of a graphical user interface, which can sometimes be tricky. A common problem is that you try to draw something, but it never appears on the screen. Here are some possible reasons why.

Wrong place: what's the origin of the coordinate system? What's the scale? Where is the component located in its parent?

Wrong size: if a component has zero width and zero height, it will be completely invisible no matter what it tries to draw— everything will be clipped. **Zero width and zero height tend to be the defaults for primitive components!** If you make a div or a span with nothing in it, it'll be zero width and height. You have to give it content, or manually set its size, to make it more reasonable size. Check whether the component (and its ancestors) have nonzero sizes.

Wrong color: is the drawing using the same color as the background? Is it using 100% alpha, so that it's completely transparent?

Wrong z-order: is something else drawing on top?

Summary

- Object, stroke, pixel approaches
- Object approach
 - CSS properties
 - Automatic redraw and double-buffering
- Stroke approach
 - Drawing contexts
 - HTML canvas
- Pixel approach
 - Alpha transparency
 - Image formats