User Interface Design and Implementation

Prof. Robert Miller

2004

Course Description

"*User Interface Design and Implementation*" introduces the principles of user interface development, focusing on three key areas:

- **Design**: How to design good user interfaces, starting with human capabilities (including the human information processor model, perception, motor skills, color, attention, and errors) and using those capabilities to drive design techniques: task analysis, user-centered design, iterative design, usability guidelines, interaction styles, and graphic design principles.

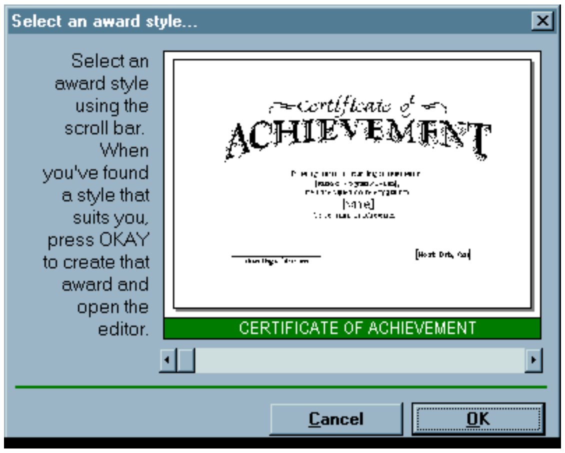
- **Implementation**: Techniques for building user interfaces, including low-fidelity prototypes, Wizard of Oz, and other prototyping tools; input models, output models, model-view-controller, layout, constraints, and toolkits.

- **Evaluation**: Techniques for evaluating and measuring interface usability, including heuristic evaluation, predictive evaluation, and user testing.

dvduc-2010

Lecture 1: Introduction to Usability

1.1 User Interface Hall of Shame



Select an award style using the scroll bar. When you've found a style that suits you, press OKAY to create that award and open the editor.

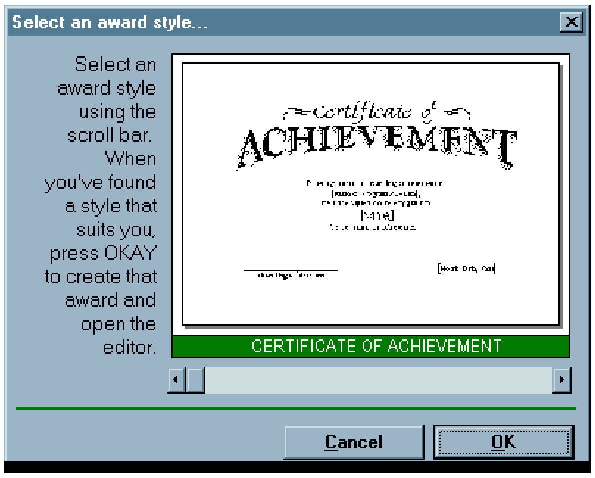
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template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

1.2 User Interface Hall of Shame



Select an award style using the scroll bar. When you've found a style that suits you, press OKAY to create that award and open the editor.

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.

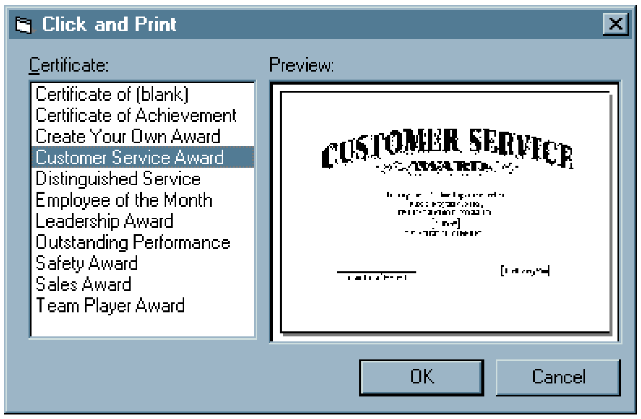
But the dialog doesn't get any better for frequent users, either. If a frequent user wants a template they've used before, how can they find it? Surely they'll remember that it's 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes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

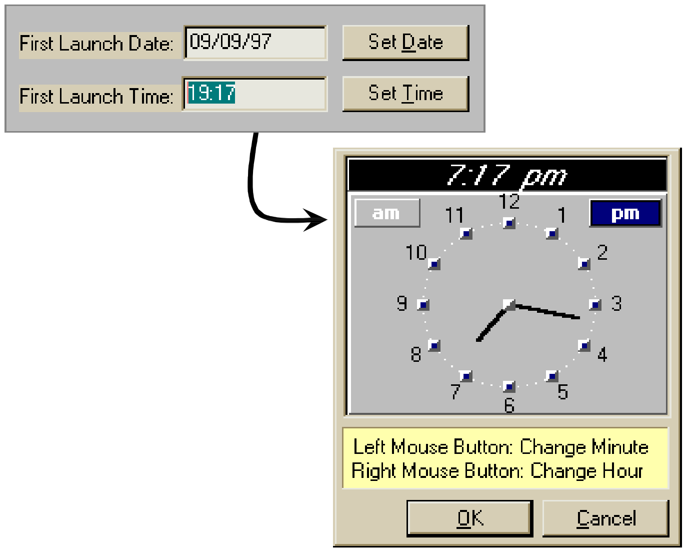
How could this dialog box be redesigned to solve some of these problems?

1.3 The Example, Redesigned



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.

1.4 More UI Hall of Shame



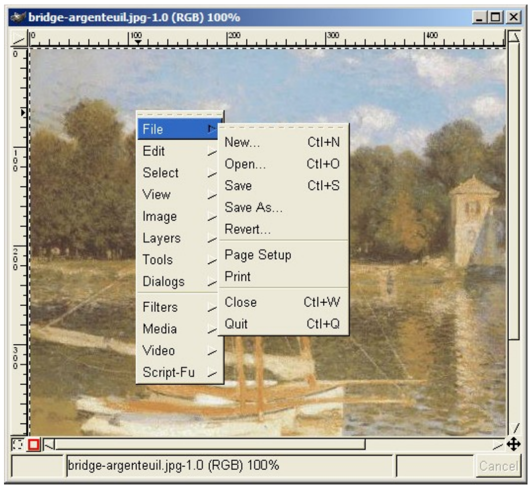
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time look like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to bring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

1.5 Hall of Shame or Hall of Fame?



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.

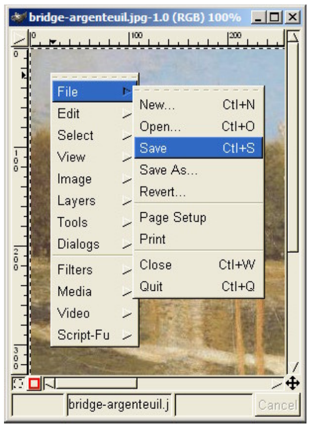
This is certainly inconsistent with other applications, and new users are likely to stumble trying to find, for example, the File menu, which never appears on a context menu in other applications. (I certainly stumbled as a new user of Gimp.) But Gimp's designers were probably thinking about expert users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

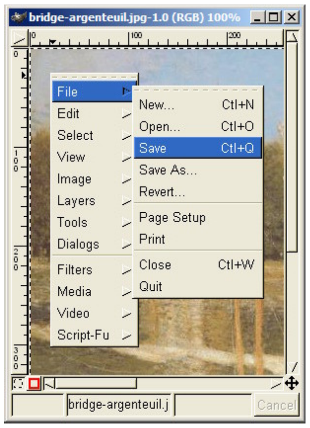
Wrong. With Gimp's design, as soon as the mouse hovers over a choice on the context menu (like File or Edit), the submenu immediately pops up to the right. That means, if I want to reach an option on the File menu, I have to move my mouse carefully to the right, staying within the File choice, until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

1.6 UI Hall of Shame or Hall of Fame?



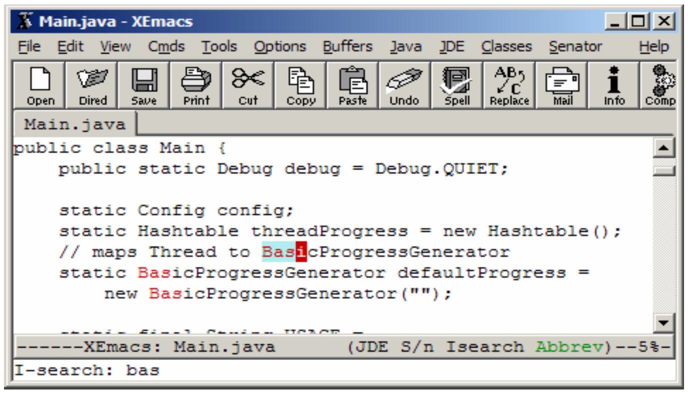


There's another interesting design feature in Gimp's menus --- well-intentioned and clever, but problematic in practice. Suppose my mouse is halfway down the File menu when I notice that the Quit command actually has a keyboard shortcut: Ctrl-Q. Great! So I press it. But it doesn't invoke the Quit command. 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 user action that has a different meaning in one state of the program than in another. Modes are 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 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 too easy to hang ourselves.

1.7 UI Hall of Shame or Hall of Fame?


Main.java - XEmacs
File Edit View Cmds Tools Options Buffers Java JDE Classes Senator Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Comp
Main.java

```
public class Main {
    public static Debug debug = Debug.QUIET;

    static Config config;
    static Hashtable threadProgress = new Hashtable();
    // maps Thread to BasicProgressGenerator
    static BasicProgressGenerator defaultProgress =
        new BasicProgressGenerator("");
------XEmacs: Main.java        (JDE S/n Isearch Abbrev)--5%-
I-search: bas
```

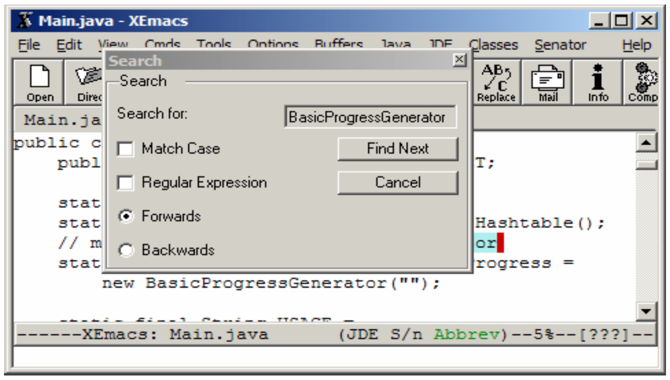
In Emacs, Ctrl-S starts an incremental search. This is a well-designed feature.

- 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?)

1.8 UI Hall of Shame


Main.java - XEmacs
File Edit View Cmds Tools Options Buffers Java JDE Classes Senator Help
Open Dired Replace Mail Info Comp
Search
Find Next
Cancel

Search for: BasicProgressGenerator
[ ] Match Case
[ ] Regular Expression
(•) Forwards
( ) Backwards

```
public c
    publ                                         T;

    stat
    stat                                         Hashtable();
    // m                                         or
    stat                                         rogress =
        new BasicProgressGenerator("");
------XEmacs: Main.java        (JDE S/n Abbrev)--5%--[???]--
```

XEmacs has menus now (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!

1.9 The User Interface Is Important

- User interface strongly affects perception of software
  - Usable software sells better

- Unusable web sites are abandoned
- Perception is sometimes superficial
  - Users blame themselves for UI failings
  - People who make buying decisions are not always end-users

So what? Why should we care about usability? After all, human beings are capable of extraordinary learning and adaptation. Even the worst interface can be fixed by a man page, right?

Putting aside the essential inhumanity of this position, there are some practical reasons why we should care about the user interfaces of our software. Usability strongly affects how software is perceived, because the user interface is the means by which the software presents itself to the world. “Ease of use” ratings appear in magazine reviews, affect word- of-mouth recommendations, and influence buying decisions. Usable software sells. Conversely, unusable software doesn’t sell. If a web site is so unusable that shoppers can’t find what they want, or can’t make it through the checkout process, then they will go somewhere else.

Unfortunately, a user’s perception of software usability is often superficial. An attractive user interface may seem “user friendly” even if it’s not really usable. Part of that is because users often blame themselves for errors they make, even if the errors could have been prevented by better interface design. (“Oops, I missed the File menu again! How stupid of me.”) So usability is a little different from other important attributes of software, like reliability, performance, or security. If the program is slow, or crashes, or gets hacked, we know who to blame. If it’s unusable, but not fatally so, the usability problems may go unreported.

1.10 The Cost of Getting It Wrong

- Users′ time isn′t getting cheaper
- Design it correctly now, or pay for it later
- Disasters happen
  - Therac-25 radiation therapy machine
  - Aegis radar system in USS Vincennes

Users don’t obey Moore’s Law. Their time doesn’t get cheaper with every new generation, like processors do. In fact, user time is probably getting more expensive every year. Interfaces that waste user time repeatedly over a lifetime of use impose a hidden cost that companies are less and less inclined to pay. For some applications, like customer call centers, saving a few seconds per call may translate into millions of dollars saved per year.

Even for shrink-wrapped software, bad user interfaces can have costs after the sale. For many software companies, a single customer support call can wipe out all the profit on that sale.

Bad user interface design can also cost lives. The Therac-25 was a radiation therapy machine for treating cancer patients. It had an electron beam with two settings: a low-energy mode, beamed directly onto the patient, and a high-energy mode in which the beam was blocked by an X-ray generating filter. Tragically, the system’s design had a race condition between the user interface and the beam controller. If the operator chose a mode, and the machine started configuring itself, and then the operator backed up and made a different choice within the 8-second interval it took for the machine to swing its magnets into place, then part of the system wouldn’t receive the new setting. As a result, a fast, experienced operator could inadvertently give severe overdoses, and several patients died. (Nancy Leveson, “Medical Devices: the Therac-25”, 1995, http://sunnyday.mit.edu/therac-25.html)

In 1988, the USS Vincennes guided missile cruiser shot down an Iranian airliner over the Persian Gulf with almost 300 people aboard. There were two failures in this incident. The radar operator interpreted the airliner as an F-14, descending as if to attack, rather than (in reality) a civilian plane that was climbing after takeoff. Both failures seemed to be caused by user interface. The IFF system was reporting the signal from an F14 on the ground at an airport hundreds of miles away, not the signal from the airliner; and the plane’s altitude readout showed only its current altitude, not the direction of change in altitude, leaving to the operator the mental comparison and calculation to determine whether the altitude was going up or down. (Peter Neumann, "Aegis, Vincennes, and the Iranian Airbus", Risks v8 n74, May 1989).

1.11 User Interfaces Are Hard to Design

- You are not the user
  - Most software engineering is about communicating with other programmers
  - UI is about communicating with users
- The user is always right
  - Consistent problems are the system′s fault
- ...but the user is not always right
  - Users aren′t designers

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!

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).

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 sUSBCxÀ „ A Ad (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.

1.12 User Interfaces are Hard to Build

- User interface takes a lot of software development effort
- UI accounts for ~50% of:
  - Design time
  - Implementation time
  - Maintenance time
  - Code size

The user interface also consumes a significant portion of software development resources. One survey of 74 software projects found that user interface code accounted for about half of the time put towards design, implementation, and maintenance, and constituted about half the code (Myers & Rosson, "Survey on user interface programming", CHI '92).

So UI is an important part of software design.

1.13 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?
  - Memorability: is it easy to remember what you learned?
  - Errors: are errors few and recoverable?
  - Satisfaction: is it enjoyable to use?

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, memorability, error rate/severity, and subjective satisfaction.

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 memorable than interface Y for some set of tasks and some class of users.

1.14 Usability Dimensions Vary In Importance

- Depends on the user
  - Novice users need learnability
  - Infrequent users need memorability
  - Experts need efficiency
- But no user is uniformly novice or expert
  - Domain experience
  - Application experience
  - Feature experience

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.

1.15 Usability Is Only One Attribute of a System

- Software designers have a lot to worry about:

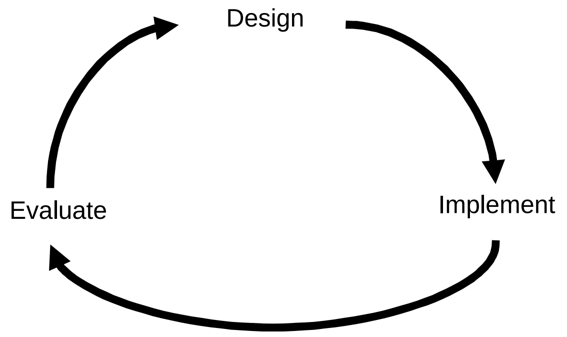
– Functionality	– **Usability**
– Performance	– Size
– Cost	– Reliability
– Security	– Standards

- Many design decisions involve tradeoffs among different attributes
- We'll take an extreme position in this class

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.

1.16 Usability Engineering Is a Process



So how do we engineer usability into our systems? By means of a process, with careful attention to detail. One goal of this course is to give you experience with this usability engineering process.

1.17 Design

- Task analysis
  - "Know thy user"
- Design guidelines
  - Avoid bonehead mistakes
  - May be vague or contradictory

The first step of usability engineering is knowing who your users are and understanding their needs. Who are they? What do they already know? What is their environment like? What are their goals? What information do they need, what steps are involved in achieving those goals? These are the ingredients of a task analysis. Often, a task analysis needs to be performed in the field – interviewing real users and watching them do real tasks.

Design guidelines are an important part of usability engineering. We've already mentioned a few: consistency, affordances, shortcuts, preventing errors. Design guidelines help you

get started, and help avoid the most boneheaded mistakes like the ones we saw at the beginning of this lecture. But guidelines are heuristics, not hard-and-fast rules. An interface cannot be made usable merely by slavish adherence to design guidelines, because guidelines may be vague or contradictory. The user should be in control, says one guideline, but at the same time we have to prevent errors. Another dictates that the user should always know what's happening, but don't forget to keep the user mental workload within acceptable limits. Design guidelines help us discuss design alternatives sensibly, but they don't give all the answers.

1.18 Implement

- Prototyping
  - Cheap, throw-away implementations
  - Low-fidelity: paper, Wizard of Oz
  - Medium-fidelity: HTML, Visual Basic
- GUI implementation techniques

  - Input/output models
  - Toolkits
  - UI builders

Since we can't predict usability in advance, we build prototypes – the cheaper, the better. We want to throw them away. We'll see that paper is a great prototyping tool! Eventually, however, the prototypes must give way to an actual, interactive computer system. A range of techniques for structuring graphical user interfaces have been

developed. We'll look at how they work, how to use them, and how UI toolkits themselves are implemented.

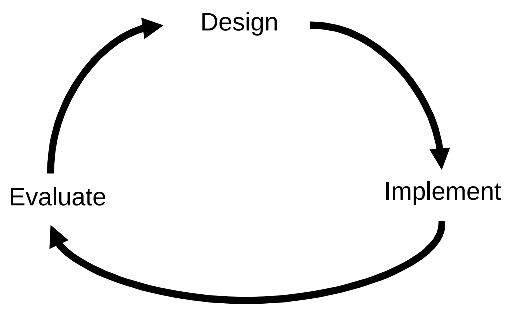
1.19 Evaluate

- Evaluation puts prototypes to the test
- Expert evaluation
  - Heuristics and walkthroughs
- Predictive evaluation
  - Testing against an engineering model (simulated user)
- Empirical evaluation
  - Watching users do it

Then we evaluate our prototypes – sometimes using heuristics, but more often against real users. Putting an implementation to the test is the only real way to measure usability.

1.20 Iterative Design

- Rinse, lather, repeat!



The most important element of usability engineering is iterative design. That means we don't go around the design-implement-evaluate loop just once. We 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.

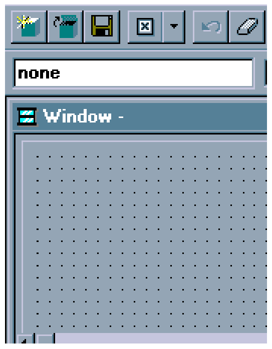
Many of the techniques we'll learn in this course are optimizations for the iterative design process: design guidelines reduce the number of iterations by helping us make better designs; cheap prototypes and discount evaluation techniques reduce the cost of each iteration. But even more important than 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.

1.21 Goals of The Course

- In this course, you'll learn:
  - how to discover users' tasks
  - how human capabilities influence usability
  - guidelines for good UI design
  - the importance of iterative design
  - how to build cheap prototypes
  - techniques for UI implementation
  - how to evaluate UIs
    - Expert (heuristic) evaluation
    - User testing
  - current research in user interfaces

Lecture 2: User-Centered Design

2.1 UI Hall of Fame or Shame?



Sybase PowerBuilder is an application development environment, not unlike Microsoft Visual Basic. Users of PowerBuilder construct forms by drawing controls (buttons, listboxes, graphical objects) on the form.

Controls are selected by pulling down a menu from a toolbar icon. The menu actually looks like a palette, but it behaves like a pulldown menu in that once you make a selection, it disappears. After a control is selected, its icon is shown in the toolbar, and clicking on the form drops the control where you clicked.

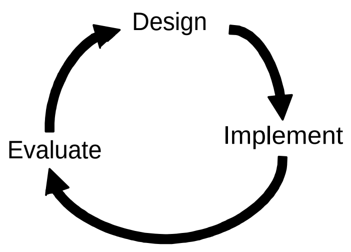
This design solves some interesting problems. Most of the time, the palette is hidden, saving screen real estate that the user might prefer to use to view the form being created. It makes it easy to drop multiple instances of the same kind of control on the form – an array of textboxes, for example, or several command buttons. The current palette mode is displayed even when the palette isn't visible.

But that last feature leads to the unfortunate problem with this design: the toolbar icon is different every time the user tries to find it! Even frequent PowerBuilder users report the disconcerting feeling of hunting around for this button. The button is always in the same place, but that doesn't make it easy to find, since it's located in the midst of other toolbar buttons. Shape is the best discriminator here, but the icon keeps changing shape.

A task that probably seemed trivial to PowerBuilder's developers – the user must know where that button was, since they've already used it! – turns out not to be trivial at all.

2.2 Today's Topics

- Iterative Design



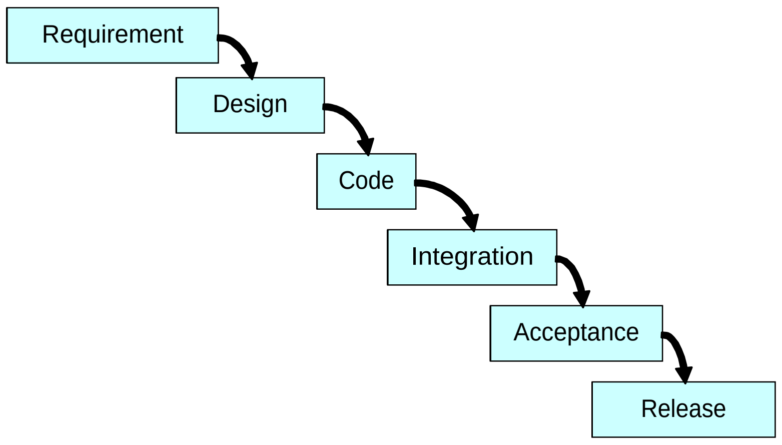
- Task Analysis

Today's lecture concerns two topics.

First, we'll look at UI design from a very high-level, considering the shape of the process that we should use to build user interfaces. **Iterative design** is the current best-practice process for developing user interfaces. It's a specialization of the spiral model described by Boehm for general software engineering. Your term project is structured as an iterative design.

Second, we'll look at how to get started with UI design – how to start the crank and get the UI design cycle going. Task analysis is the process by which you discover the characteristics of your target users and the tasks that they need to solve.

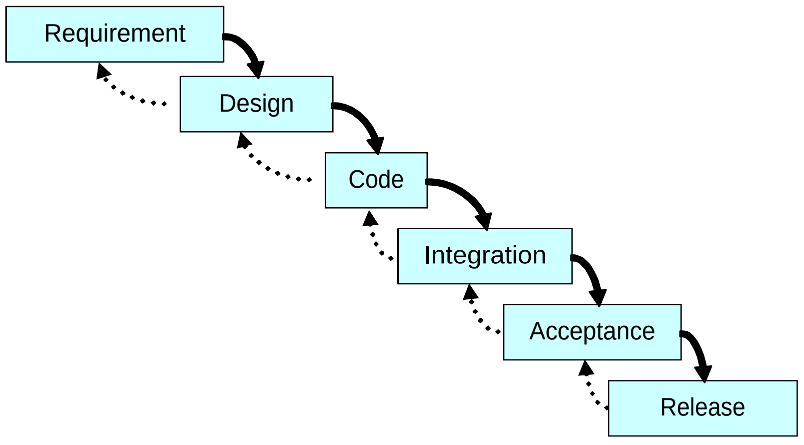
2.3 Traditional Software Engineering Process: Waterfall Model



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 lecturers handed you a set of requirements for the software you had to build --- e.g. the specification of GizmoBall or Antichess. (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.

2.4 Feedback in the Waterfall Model



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!)

2.5 Waterfall Model Is Bad for UI Design

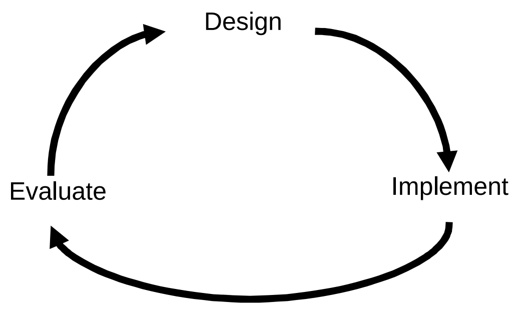
- User interface design is risky
  - So we’re likely to get it wrong
- Users are not involved in validation until acceptance testing
  - So we won’t find out until the end
- UI flaws often cause changes in requirements and design
  - So we have to throw away carefully-written and tested code

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 last lecture that slapping on patches doesn’t fix serious usability problems.

2.6 Iterative Design



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).

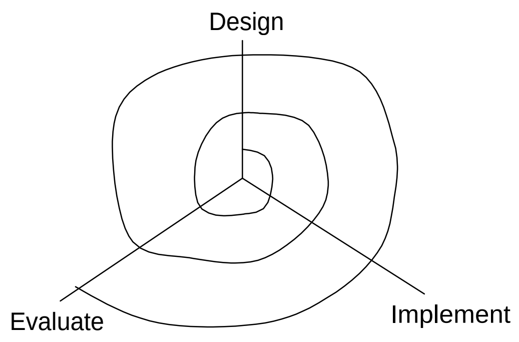
OK – but this just looks like the worst-case waterfall model, where we made it all the way from design to acceptance testing before discovering a design flaw that forced us to repeat the process! What’s the trick here?

2.7 Iterative Design the Wrong Way

- Every iteration corresponds to a release
  - Evaluation (complaints) feeds back into next version’s design
- Using your paying customers to evaluate your usability
  - They won’t like it
  - They won’t buy version 2

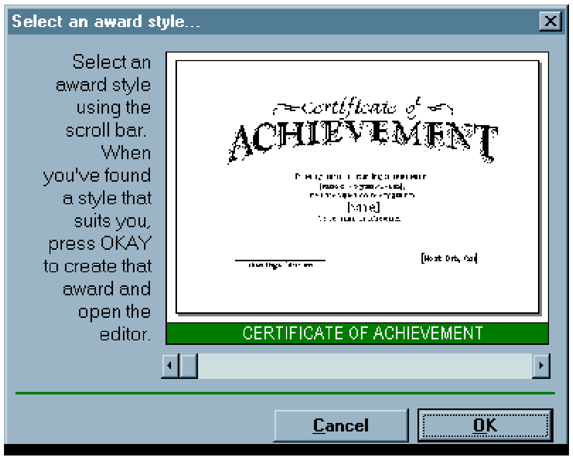
Unfortunately, many commercial UI projects have followed this model. They do a standard waterfall design, produce a bad UI, and release it. Evaluation then takes place in the marketplace, as hapless customers buy their product and complain about it. Then they iterate the design process on version 2.

2.8 Spiral Model



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.

2.9 Early Prototypes Can Detect Usability Problems


Select an award style...

Select an award style using the scroll bar. When you've found a style that suits you, press OKAY to create that award and open the editor.

Remember this Hall of Shame candidate from last 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's code.

2.10 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

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.

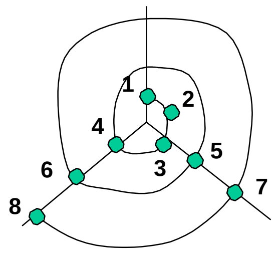
2.11 User-Centered Design

- Iterative design
- Early focus on users and tasks
  - user analysis: who the users are
  - task analysis: what they need to do
  - involving users as evaluators, consultants, and sometimes designers
- Constant evaluation
  - Users are involved in every iteration
  - Every prototype is evaluated somehow

Iterative design is a crucial part of **user-centered design**, the design process for user interfaces that is widely accepted among UI practicioners. A variety of brand-name user- centered design techniques exist (e.g., GUIDE, STUDIO, OVID, LUCID). But most have three features in common:

- iterative design using rapid prototyping
- early focus on users and tasks
- evaluation throughout the iterative design process.

2.12 User-Centered Design in Software Engineering Course



1. Task analysis
2. Design sketches
3. Paper prototype
4. In-class user testing
5. Computer prototype
6. Heuristic evaluation
7. Implementation
8. User testing

The term project's milestones are designed to follow a spiral model:

1. task analysis (1 week): collecting the requirements for the UI, which we'll discuss in the second half of this lecture.
2. design sketches (1 week): paper sketches of your UI design
3. paper prototype (2 week): an interactive prototype made of paper and other cheap physical materials
4. in-class user testing (1 day): one day during the paper prototype assignment, we'll spend the lecture using each others' prototypes
5. computer prototype (2 weeks): an interactive software prototype that you plan to throw away
6. heuristic evaluation (1 week): we'll exchange software prototypes and evaluate them as usability experts would
7. 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 did in a software engineering course – step 7 – is only one milestone in this class!

2.13 Case Study: 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

The Olympic Message System is a good 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.

2.14 User & Task Analysis

- First step of user-centered design
- User analysis: who is the user?
- Task analysis: what does the user need to do?

We've seen that UI design is iterative – that we have to turn the crank several times to achieve good usability. How do we get started? How do we acquire information for the initial design?

The process of collecting information for the first design is called user and task analysis. In these steps, we ask who the users are, and what it is that they're trying to accomplish.

2.15 Know Thy User

- Identify characteristics of target user population
  - Age, gender, ethnicity
  - Education
  - Physical abilities
  - General computer experience
  - Skills (typing? reading?)
  - Domain experience
  - Application experience

  - Work environment and other social context
  - Relationships and communication patterns

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?

2.16 Multiple Classes of Users

- Many applications have several kinds of users
- Example: Olympic Message System
  - Athletes
  - Friends & family
  - Telephone operators
  - Sysadmins

Many, if not most, applications have to worry about multiple classes of users. Do a user analysis for every class.

2.17 How To Do User Analysis

- Techniques
  - Questionnaires
  - Interviews
  - Observation
- Obstacles
  - Developers and users may be systematically isolated from each other
    - Tech support shields developers from users
    - Marketing shields users from developers
  - Some users are expensive to talk to
    - Doctors, executives, union members

The best way to do user analysis is to find some representative users and ask them. Straightforward characteristics can be obtained by a questionnaire. Details about context and environment can be obtained by interviewing users directly, or even better, observing them going about their business, in their natural habitat.

Sometimes it can be hard to reach users. Software companies can erect artificial barriers between users and developers, for their mutual protection. After all, if users know who the developers are, they might pester them with bugs and questions about the software, which are better handled by tech support personnel. The marketing department may be afraid to let the developers interact with the users – not only because geeks can be scary, but also because usability discussions may make customers dissatisfied with the current product. ("I hadn't noticed it before, but that DOES suck!")

Some users are also expensive to find and talk to. Nevertheless, make every effort to collect the information you need. A little money spent collecting information initially should pay off significantly in better designs and fewer iterations.

2.18 Example: Self-Service Grocery Checkout

- Who are the users?
  - Grocery shoppers
  - Wide range of ages (10-80) and physical abilities (height, mobility, strength)
  - No computer experience
  - No training: walk up and use
  - Knowledge of food, but not about supermarket inventory techniques
  - Supermarket shoppers often ask each other for help finding things
- Major user classes
  - Family shopping is often done by women, often accompanied by small children
  - Store clerks who need to help shoppers

Let's look at an example. Suppose we've been charged with designing a system that will allow grocery shopper to ring up and pay for their purchases themselves. This slide shows some of the observations that might be included in a user analysis.

Class made some good points here:

What supermarket? Upscale vs. downscale

Quick stoppers

Family shoppers (possibly with kids in tow)

Quick stoppers

- don't use coupons
- possibly illiterate, although they may select themselves out

2.19 Task Analysis

- Identify the individual tasks the program might solve
- Each task is a goal (what, not how)
- Often helps to start with overall goal of the system and then decompose it hierarchically into tasks
  - Overall goal: shoppers pay for their own groceries
  - Tasks:
    - Enter groceries into register
    - Bag groceries
    - Pay

The next step is figuring out what tasks are involved in the problem. A task should be expressed as a goal: what needs to be done, not how.

One good way to get started on a task analysis is hierarchical decomposition. Think about the overall problem you're trying to solve. That's really the top-level task. Then decompose it into a set of subtasks, or subgoals, that are part of satisfying the overall goal.

2.20 Essential Parts of Task Analysis

- What needs to be done?
  - Goal
- What must be done first to make it possible?
  - Preconditions
    - Tasks on which this task depends
    - Information that must be known to the user

- What steps are involved in doing the task?
  - Subtasks
  - Subtasks may be decomposed recursively

Once you've identified a list of tasks, fill in the details on each one. Every task in a task analysis should have at least these parts.

The goal is just the name of the task, like "send an email message."

The preconditions are the conditions that must be satisfied before it's reasonable or possible to attempt the task. Some preconditions are other tasks in your analysis; in the grocery checkout example, entering all the groceries is a precondition to paying. Other preconditions are information needs, things the user needs to know in order to do the task. For example, in order to send an email message, I need to know the email addresses of the people I want to send it to; I may also need to look at the message I'm replying to. Preconditions are vitally important to good UI design, particularly because users don't always satisfy them before attempting a task, resulting in errors. Knowing what the preconditions are can help you prevent these errors, or at least render them harmless. For example, a precondition of starting a fire in a fireplace is opening the flue, so that smoke escapes up the chimney instead of filling the room. If you know this precondition as a designer, you can design the fireplace with an interlock that ensures the precondition will be met. Another design solution is to offer opportunities to complete preconditions: for example, an email composition window should give the user access to their address book to look up recipients' email addresses.

Finally, decompose the task into subtasks, individual steps involved in doing the task. If the subtasks are nontrivial, they can be recursively decomposed in the same manner.

2.21 Example: Self-service Grocery Checkout

- Goal
  - Enter groceries into register
- Preconditions
  - All the groceries you want are in your cart
- Subtasks
  - Enter prepackaged item
  - Enter loose produce

Here's the continuation of our grocery cart example, looking at the first task we identified.

2.22 Other Questions to Ask About a Task

- Where is the task performed?
  - Front of supermarket, standing up
- How often is the task performed?
  - At most a few times a week
- What are its time or resource constraints?
  - A minute or two
- How is the task learned?
  - By trying it
  - By watching others
  - By being shown how by store personnel
- What can go wrong? (Exceptions, errors, emergencies)
  - Barcode is missing or smudged
  - Shopper wants to buy alcohol or cigarettes
- Who else is involved in the task?

There are lots of questions you should ask about each task. Here are a few, with examples from the grocery checkout.

2.23 How to Do a Task Analysis

- Interviews with users
- Direct observation of users performing tasks

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 grocery checkout example, we would want to observe store cashiers checking out groceries in order to understand the grocery checkout task. We would also want to interview grocery store shoppers, in order to understand better their goals in the task.

2.24 Dangers of Task Analysis

- Duplicating a bad existing procedure in software
- Failing to capture good aspects of existing procedure

One danger of task analysis derived from observation is that it may give too much weight to the way things are currently done, even if they're inefficient or could be done completely differently 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, not just what they do.

Conversely, 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 important.

2.25 Hints for Better User & Task Analysis

- Questions to ask
  - Why do you do this? (goal)
  - How do you do it? (subtasks)
- Look for weaknesses in current situation
  - Goal failures, wasted time, user irritation
- Contextual inquiry
- Participatory design

When you're interviewing users, they tend to focus on the what: "first I do this, then I do this…" Be sure to probe for the why and how as well, to make your analysis more abstract and at the same time more detailed.

Since you want to improve the current situation, look for its weaknesses and problems. What tasks often fail? What unimportant tasks are wasting lots of time? It helps to ask the users what annoys them and what suggestions they have for improvement.

There are two other techniques for making user and task analysis more effective: contextual inquiry and participatory design.

2.26 Contextual Inquiry

- Observe users doing real work in the real work environment
- Be concrete
- Establish a master-apprentice relationship
  - User shows how and talks about it
  - Interviewer watches and asks questions

- Challenge assumptions and probe surprises

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)

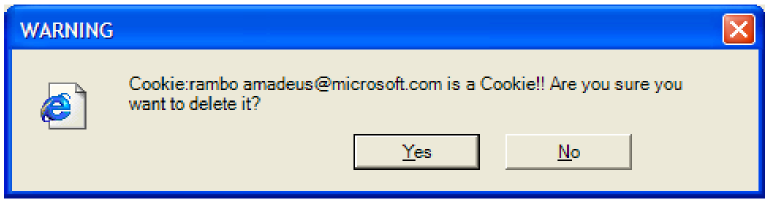
2.27 Participatory Design

- Include representative users directly in the design team
- OMS design team included an Olympic athlete as a consultant

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.

Lecture 3: UI Software Architecture

3.1 UI Hall of Fame or Shame?



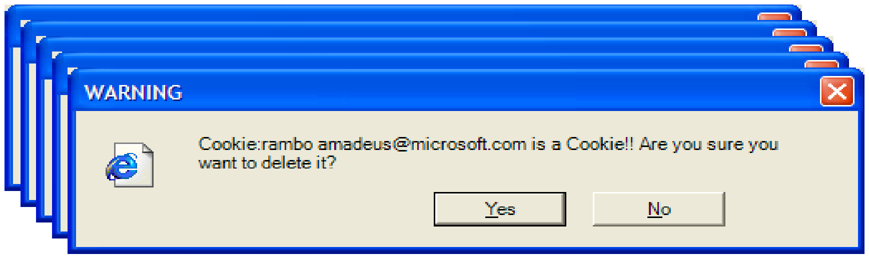
WARNING

Cookie:rambo amadeus@microsoft.com is a Cookie!! Are you sure you want to delete it?

This message used to appear when you tried to delete the contents of your Internet Explorer cache from within the Windows Explorer.

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? What's a Cookie? What will happen if I delete it? Don't ask questions the user can't answer.

3.2 Hall of Shame



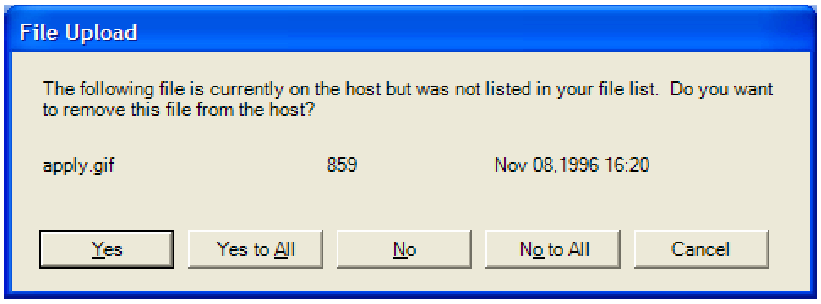
WARNING

Cookie:rambo amadeus@microsoft.com is a Cookie!! Are you sure you want to delete it?

And definitely don't ask more than once. There may be hundreds of cookies cached in the browser; this dialog box appears for each one the user selected.

There's something missing from the dialog, whose absence becomes acute once the dialog appears a few times: a Cancel button. Always give users a way to **escape**.

3.3 UI Hall of Fame or Shame?



File Upload

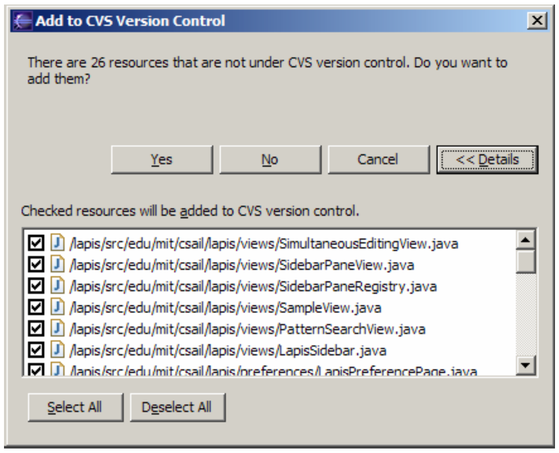
The following file is currently on the host but was not listed in your file list. Do you want to remove this file from the host?

apply.gif 859 Nov 08,1996 16:20

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**, 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? You'd have to say No to every dialog until you found that file.

3.4 Hall of Fame



There are 26 resources that are not under CVS version control. Do you want to add them?

Checked resources will be added to CVS version control.

```
/lapis/src/edu/mit/csail/lapis/views/SimultaneousEditingView.java
/lapis/src/edu/mit/csail/lapis/views/SidebarPaneView.java
/lapis/src/edu/mit/csail/lapis/views/SidebarPaneRegistry.java
/lapis/src/edu/mit/csail/lapis/views/SampleView.java
/lapis/src/edu/mit/csail/lapis/views/PatternSearchView.java
/lapis/src/edu/mit/csail/lapis/views/LapisSidebar.java
/lapis/src/edu/mit/csail/lapis/preferences/LapisPreferencePage.java
```

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, provided by IBM 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.

Nevertheless, this deserves to be in the hall of fame.

3.5 Today's Topics

- Model-view-controller
- View hierarchy
- Observer

Starting with today's lecture, we'll be talking 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 hierarchy, which is a central feature in the architecture of every popular GUI toolkit; and the observer pattern, which is essential to decoupling the model from the view and controller.

3.6 Model-View-Controller Pattern

- Separates frontend concerns from backend concerns
- Separates input from output
- Permits multiple views on the same application data
- Permits views/controllers to be reused for other models
- Example: text box
  - Model: mutable string
  - View: rectangle with text drawn in it
  - Controller: keystroke handler

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).

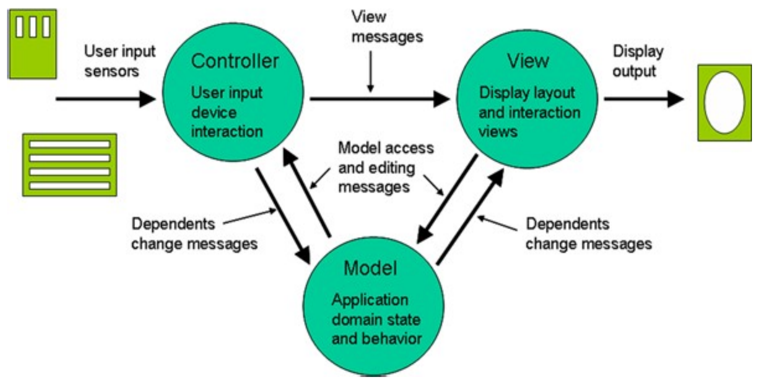
MVC's primary goal is separation of concerns. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is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

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 controllers to be reused for other models, in other applications. The MVC pattern enables the creation of user interface toolkits, which are libraries of reusable interface objects.

In practice, the MVC pattern doesn't quite work out the way we'd like. We'll see why.

A simple example of the MVC pattern is a text box 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.

3.7 MVC Diagram


User input sensors
Controller
User input device interaction
View messages
View
Display layout and interaction views
Display output
Model access and editing messages
Dependents change messages
Dependents change messages
Model
Application domain state and behavior

Here's a schematic diagram of the interactions between model, view, and controller.

3.8 Model

- Responsible for data
  - Maintains application state (data fields)
  - Implements state-changing behavior
  - Notifies dependent views/controllers when changes occur (observer pattern)
- Design issues
  - How fine-grained are the change descriptions?
    - "The string has changed somehow" vs. "Insertion between offsets 3 and 5"
  - How fine-grained are the observable parts?
    - Entire string vs. only the part visible in a view

Let's look at each part in a little more detail. 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 **observer pattern**, in which interested views and controllers register themselves as listeners for events generated by the model.

Designing these notifications is not always trivial, because a model typically has many parts that might have changed. Even in our simple text box example, the string model has a number of characters. A list box has a list of items. When a model notifies its views about a change, how finely should the change be described? Should it simply say “something has changed”, or should it say “these particular parts have changed”? Fine-grained notifications may save dependent views from unnecessarily querying state that hasn’t changed, at the cost of more bookkeeping on the model’s part.

Fine-grained notifications can be taken a step further by allowing views to make fine-grained registrations, registering interest only in certain parts of the model. Then a view displaying a small portion of a large model would only receive events for changes in the part it’s interested in. Reducing the grain of notification or registration is crucial to achieving good interactive view performance on large models.

3.9 View

- Responsible for output
  - Occupies screen extent (position, size)
  - Draws on the screen
  - Listens for changes to the model
  - Queries the model to draw it
- A view has only one model
  - But a model can have many views

In MVC, view objects are responsible for output. A view 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.

3.10 Controller

- Responsible for input
  - Listens for keyboard & mouse events
  - Instructs the model or the view to change accordingly
    - e.g., character is inserted into the text string
- A controller has only one model and one view

Finally, the controller handles all the input. It receives keyboard and mouse events, and instructs the model to change accordingly. For example, the controller of a text box receives keystrokes and inserts them into the text string.

In the original MVC pattern used in Smalltalk-80, there was only one controller for each model and view.

3.11 Problem: Controller Needs Output

- Menus are clearly controller-related
  - e.g. right-click menu on a text field
- But a menu needs to be drawn
  - A menu is a model-view-controller in itself, used as a subcomponent

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. A good example is a popup menu – in the context of our text box example, this might be the right-click menu that lets you cut, copy, or paste. The menu is clearly part of the controller. Its appearance depends on the controller’s state -- e.g., highlighting the menu option that the mouse is hovering over – not strictly on the model’s state, like the view does.

3.12 Problem: Who Manages Selection?

- Must be displayed by the view

– As blinking text cursor or highlighted object
- Must be updated and used by the controller
  - – Clicking or arrow keys change selection
  - – Commands modify the model parts that are selected
- 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)

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.

3.13 Problem: Direct Manipulation

- Direct manipulation: user points at displayed objects and manipulates them directly
- View must provide affordances for controller
  - – e.g. scrollbar thumb, selection handles
- View must also provide feedback about controller state
  - – e.g., button is depressed

Here's a third example of why input and output are hard to decouple. Good graphical user interfaces support direct manipulation, which means that the user can manipulate displayed objects directly, as if they were physical objects. A scrollbar is a good example of direct manipulation: the user can change the position of the scrollbar thumb by clicking and dragging it directly. Drawing editors provide lots of direct manipulation: you can drag an object to the position you want it, and you can drag the selection handles drawn around it to resize the object.

Direct manipulation techniques force a close cooperation between the view and the controller. The view must display affordances for manipulation, 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.

3.14 Reality: Tightly Coupled View & Controller

- MVC has largely been superseded by MV (Model-View)
- A reusable view manages both output and input
  - – Also called widget or component
- Reusable controllers are rare
  - – Actions in Java Swing: objects that sit behind menu item, toolbar button, or keyboard shortcut

- E.g. cut, copy, paste, delete

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

The term MVC still persists, but people who use it tend to be talking about a higher level of system design than the GUI. For example, MVC is used in the Java Server Pages architecture to mean a database (model), request handler (controller), and a reply page generator (view). At this level of abstraction, the controller and the view can be significantly decoupled from each other.

Are there any vestiges of independent, reusable controllers left in modern GUIs? Certainly any input event handler – like the MouseListener interface in Java – is a controller. But MouseListener is an interface, not a reusable component, and the MouseListener interface tends to be implemented by a view (or by an inner class of a view) – you can't find many reusable MouseListener implementations sitting in the Java library waiting to be added to your program.

Reusable controllers do still exist, but at a higher level than raw mouse and keyboard input. In Java Swing, for example, an Action is a reusable object that represents a command. It sits behind a menu item, toolbar button, or keyboard shortcut, and gets triggered when the user invokes it. Swing actually includes a number of reusable Actions for editing text models: cut, copy, paste, delete, etc.

3.15 View Hierarchy

- Views are arranged into a hierarchy
- Containers
  - Window, panel, rich text widget
- Components
  - Canvas, button, label, textbox
  - Containers are also components
- Every GUI system has a view hierarchy, and the hierarchy is used in lots of ways
  - Output
  - Input
  - Layout

The second important pattern we want to discuss in this lecture is the **view hierarchy**.

Views are arranged into a hierarchy of containment, which some views (called containers in the Java nomenclature) can contain other views (called components in Java). A crucial feature of this hierarchy is that containers are themselves components – i.e., Container is a subclass of Component. Thus a container can include other containers, allowing a hierarchy of arbitrary depth.

Virtually every GUI system has some kind of view hierarchy. The view hierarchy is a powerful structuring idea, which is loaded with a variety of responsibilities in a typical GUI. We'll look at three ways the view hierarchy is used: for output, input, and layout.

3.16 View Hierarchy: Output

- Drawing
  - Draw requests are passed top-down through the hierarchy
- Clipping
  - Parent container prevents its child components from drawing outside its extent
- Z-order
  - Children are (usually) drawn on top of parents
  - Child order dictates drawing order between siblings
- Coordinate system
  - Every container has its own coordinate system (origin usually at the top left)

  - Child positions are expressed in terms of parent coordinates

First, and probably primarily, the view hierarchy is used to organize output: drawing the views on the screen. Draw requests are passed down through the hierarchy. When a container is told to draw itself, it must make sure to pass the draw request down to its children as well.

The view hierarchy also enforces a spatial hierarchy by clipping – parent containers preventing their children from drawing anything outside their parent's boundaries.

The hierarchy also imposes an implicit layering of views, called z-order. When two components overlap in extent, their z-order determines which one will be drawn on top. The z-order corresponds to an in-order traversal of the hierarchy. In other words, children are drawn on top of their parents, and a child appearing later in the parent's children list is drawn on top of its earlier siblings.

Each component in the view hierarchy has its own coordinate system, with its origin (0,0) usually at the top left of its extent. The positions of a container's children are expressed in terms of the container's coordinate system, rather than in terms of full-screen coordinates. This allows a complex container to move around the screen without changing any of the coordinates of its descendents.

3.17 View Hierarchy: Input

- Event dispatch and propagation
  - Raw input events (key presses, mouse movements, mouse clicks) are sent to lowest component
  - Event propagates up the hierarchy until some component handles it
- Keyboard focus
  - One component in the hierarchy has the focus (implicitly, its ancestors do too)

In most GUI systems, the view hierarchy also participates in input handling.

Raw mouse events – button presses, button releases, and movements – are sent to the smallest component (deepest in the view hierarchy) that encloses the mouse position. If this component chooses not to handle the event, it passes it up to its parent container. The event propagates upward through the view hierarchy until a component chooses to handle it, or until it drops off the top, ignored.

Keyboard events are treated similarly, except that the first component to receive the event is determined by the keyboard focus, which always points to some component in the view hierarchy.

3.18 View Hierarchy: Layout

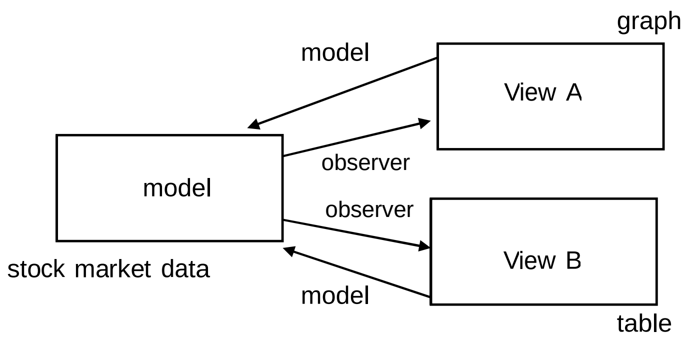
- Automatic layout: children are positioned and sized within parent
  - Allows window resizing
  - Smoothly deals with internationalization and platform differences (e.g. fonts or widget sizes)
  - Lifts burden of maintaining sizes and positions from the programmer
    - Although actually just raises the level of abstraction, because you still want to get the graphic design (alignment & spacing) right

The view hierarchy is also used to direct the layout process, which determines the extents (positions and sizes) of the views in the hierarchy. Many GUI systems have supported automatic layout, including Motif (an important early toolkit for X Windows), Tk (a toolkit developed for the Tcl scripting language), and of course Java AWT and Swing.

Automatic layout is most useful because it allows a view hierarchy to adjust itself automatically when the user resizes its window, changing the amount of screen real estate allocated to it. Automatic layout also smoothly handles variation across platforms, such as differences in fonts, or differences in label lengths due to language translation.

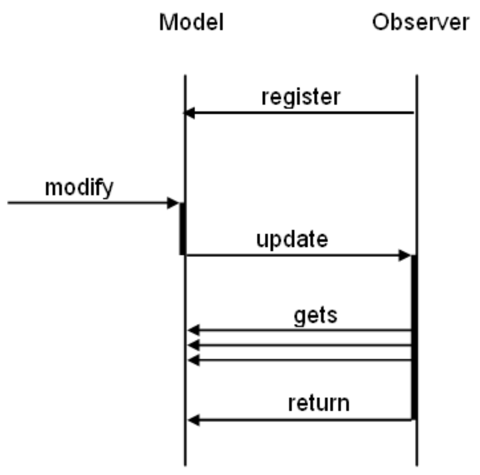
3.19 Observer Pattern

- Observer pattern is used to decouple model from views



Finally, let’s look at the **observer pattern**.

3.20 Basic Interaction



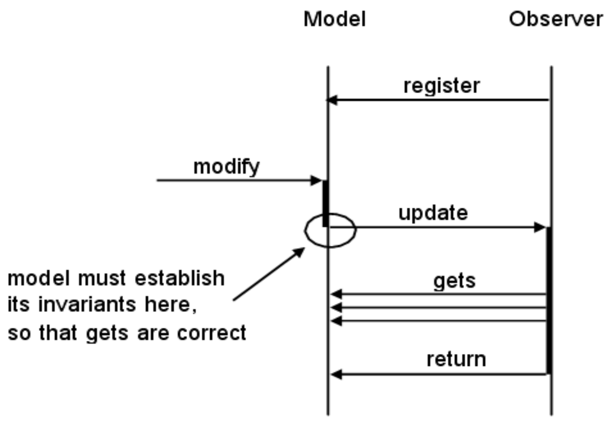
```
interface Model {
   void register(Observer)
   void unregister(Observer)
   Object get()
   void modify()
}

interface Observer {
   void update(Event)
}
```

Here’s the conventional interaction that occurs in the observer pattern. (We’ll use the abstract representation of Model and Observer shown on the right. Real models and observers will have different, more specific names for the methods, and different method signatures. They’ll also have multiple versions of each of these methods.)

1. An observer registers itself to receive notifications from the model.
2. When the model changes (usually due to some other object modifying it), the model broadcasts the change to all its registered views by calling update on them. The update call usually includes some information about what change occurred. One way is to have different update methods on the observer for each kind of change (e.g. treeStructureAdded() vs. treeStructureRemoved()). Another way is to package the change information into an event object. Regardless of how it’s packaged, this change information that is volunteered by the model is usually called pushed data.
3. An observer reacts to the change in the model, often by pulling other data from the model using get calls.

We already discussed the tradeoff between fine-grained and coarse-grained registration and notification. There’s also a tradeoff between pushing and pulling data.

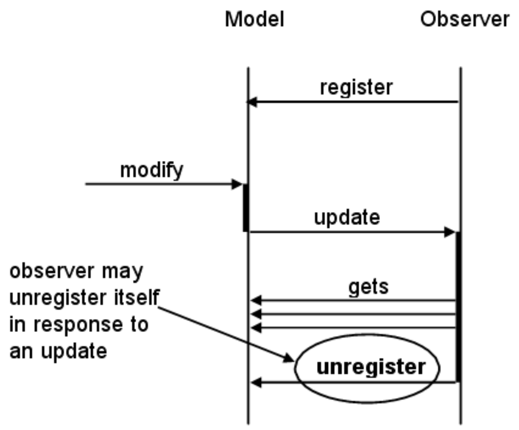
3.21 Model Must Be Consistent Before Update



Let's talk about some important issues. First, when the model calls **update** on its observers, it is giving up control – in much the same way that a method gives up control when it returns to its caller. Observers are free to call back into the model, and in fact often do in order to pull information from it. So the model has to make sure that it's consistent --- i.e., that it has established all of its internal invariants – before it starts issuing notifications to observers.

So it's often best to delay firing off your observers until the end of the method that caused the modification. Don't fire observers while you're in the midst of making changes to the model's data structure.

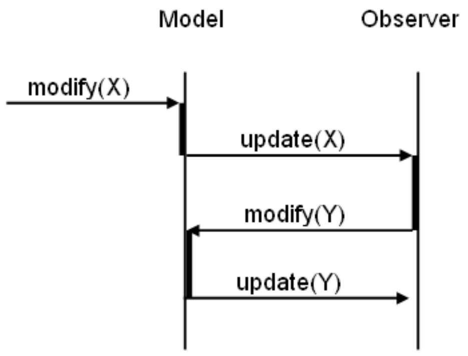
3.22 Registration Changes During Update



Another potential pitfall is observers that unregister themselves. For example, suppose the model contains stock market data, and a view registers itself as an observer of one stock in order to watch for that stock reaches a certain price. Once the stock hits the target price, the view does its thing (e.g., popping up a window to notify the user); but then it's no longer needed, so it unregisters itself from the model.

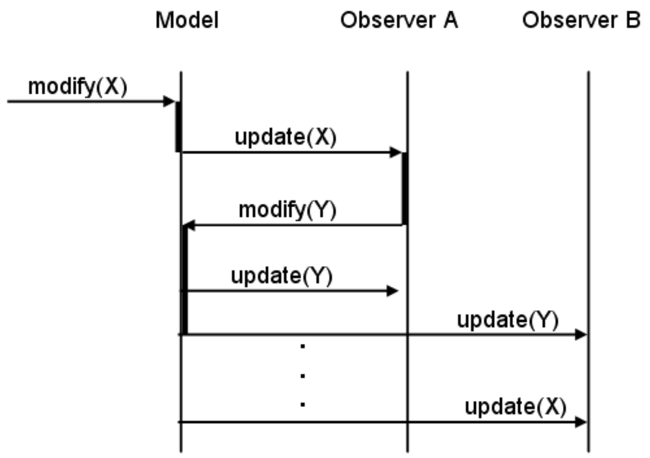
This is a problem if the model is iterating naively over its collection of observers, and the collection is allowed to change in the midst of the iteration. It's safer to iterate over a copy of the observer list. Since one-shot observers are not particularly common, however, this imposes an extra cost on every event broadcast. So the ideal solution is to copy the observer list only when necessary – i.e., when a register or unregister occurs in the midst of event dispatch.

3.23 Update Triggers A Modify



A third pitfall occurs when an observer responds to an update message by calling modify on the model. Why would it do that? It might, for instance, be trying to keep the model within some legal range. Obviously, this could lead to infinite regress if you're not careful. A good practice for models to protect themselves against sloppy views is to only send updates if a change actually occurs; if a client calls modify() but it has no actual effect on the model, then no updates should be sent.

3.24 Out-of-Order Updates



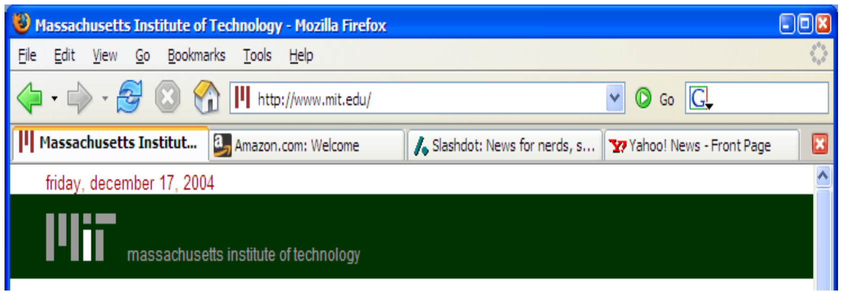
A more pernicious pitfall can arise when there are multiple observers and one of them modifies the model: events can get out of order. This diagram shows an example of what can happen. Observer A gets the first update (for change X), and it responds by modifying the model. The model in turn responds by sending out another round of updates (change Y) immediately. If observer A makes no further modifications, observer B finally gets its updates – but it gets the changes in the opposite order that they actually occurred, change Y before change X!

There are a few solutions to this problem:

- the model could delay broadcasting event Y until all the updates for X have been sent. This is usually done by putting the events on a queue. It imposes some additional cost and complexity on the model, but it's the best way to guarantee that events arrive in the same order to all observers.
- the model could skip sending the update(X) to observer B. This ensures that observer B doesn't get an event with old data in it, but it also means that B has missed a transition. Some observers might care about those transitions: for example, if B is a graph displaying stock prices over a time interval.
- observers could ignore pushed data (X, Y) and always get the latest state directly from the model. This is good practice in general. If your view only needs to show the current model state, then get it directly from the model; don't rely on the pushed event to tell you what it is.

Lecture 4: Human Capabilities

4.1 UI Hall of Fame or Shame?


Massachusetts Institute of Technology - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.mit.edu/
Go
Massachusetts Institut...
Amazon.com: Welcome
Slashdot: News for nerds, s...
Yahoo! News - Front Page
friday, december 17, 2004
massachusetts institute of technology

Today's candidate for the User Interface Hall of Fame is tabbed browsing, a feature found in almost all web browsers (Mozilla, Firefox, Safari, Konqueror, Opera) except Internet Explorer. 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.

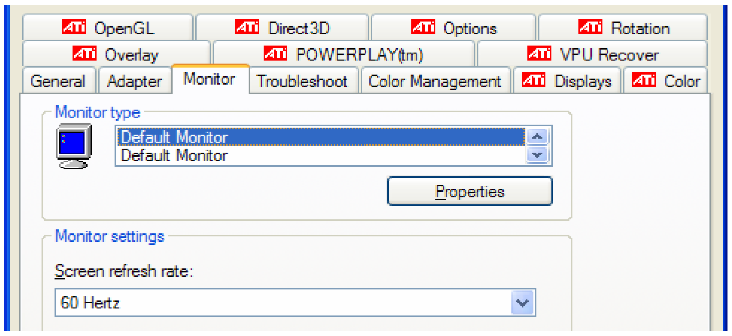
Tabbed browsing neatly solves a scaling problem in the Windows taskbar. If you accumulate several top-level Internet Explorer windows, they cease to be separately clickable buttons in the taskbar and merge together into a single Internet Explorer button with a popup menu. So your browser windows become 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. So tabbed browsing clearly wins on task analysis.

Another neat feature of tabbed browsing, at least in Mozilla, 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 would let you do this by resizing and repositioning the windows. Another problem is that, at least in Mozilla, tab groups can't be easily rearranged– moved to other windows, dragged out to start a new window.

4.2 Hall of Shame

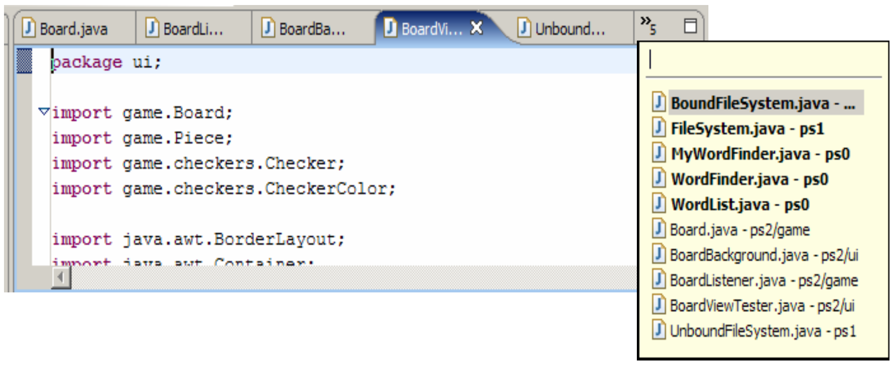


Another problem is that tabs don't really scale up either – you can't have more than 5-10 without shrinking their labels so much that they're unreadable. Some designers have tried using multiple rows of tabs, but this turns out to be a horrible idea. Here's a typical example. Clicking on a tab in a back row (like OpenGL) 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, 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. Alas, you couldn't simply replace tabs with buttons in the dialog box shown here. (Why not?) Tabbed browsing probably couldn't use buttons either, without some careful graphic design to distinguish them from bookmark buttons.

4.3 Hall of Fame or Shame?



```
package ui;

import game.Board;
import game.Piece;
import game.checkers.Checker;
import game.checkers.CheckerColor;

import java.awt.BorderLayout;
```

Here's how Eclipse 3.0 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've been using Eclipse 3.0 for months, and I only noticed this feature when I started carefully exploring the tab interface.

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 tabs are listed in the order they were last used. 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? No, because it interferes with users' spatial memory.

4.4 Today's Topics

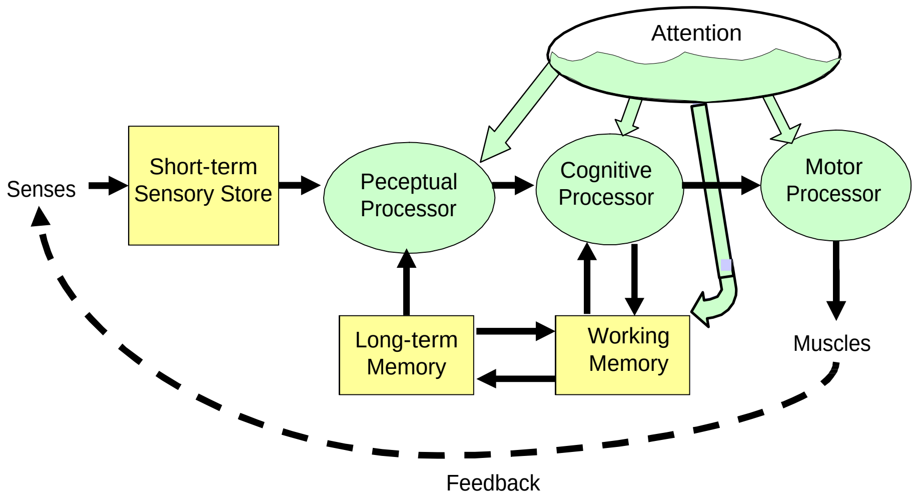
- Human information processing
  - Perception
  - Motor skills
  - Memory
  - Decision making
  - Attention
  - Vision

This course is about building effective human-computer interfaces. Just as it helps to understand the properties of the computer system you're programming for – its processor speed, memory size, hard disk, operating system, and the interaction between these components – it's going to be important for us to understand some of the properties of the human that we're designing for.

We talked last week about user analysis, which collects information about high-level properties of our target users, particularly ways in which the target users are different from ourselves.

In today's lecture, we're going to look at low-level details: the processors, memories, and properties of the human cognitive apparatus. And we will largely concentrate on properties that most of us have in common (with some important exceptions when we look at color

4.5 Human Information Processing



Here's a high-level look at the cognitive abilities of a human being -- really high level, like 30,000 feet. This is a version of the Model Human Processor was developed by Card, Moran, and Newell as a way to summarize decades of psychology research in an **engineering model**. (Card, Moran, Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, 1983) This model is different from the original MHP; I've modified it to include a component representing the human's attention resources (Wickens, *Engineering Psychology and Human Performance*, Charles E. Merrill Publishing Company, 1984).

This model is an abstraction, of course. But it's an abstraction that actually gives us *numerical parameters* describing how we behave. Just as a computer has memory and processor, so does our model of a human. Actually, the model has several different kinds of memory, and several different processors.

Input from the eyes and ears is first stored in the **short-term sensory store**. As a computer hardware analogy, this memory is like a frame buffer, storing a single frame of perception.

The **perceptual processor** takes the stored sensory input and attempts to recognize *symbols* in it: letters, words, phonemes, icons. It is aided in this recognition by the **long-term memory**, which stores the symbols you know how to recognize.

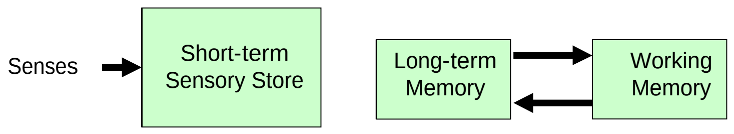
The **cognitive processor** takes the symbols recognized by the perceptual processor and makes comparisons and decisions. It might also store and fetch symbols in **working memory** (which you might think of as RAM, although it's pretty small). The cognitive processor does most of the work that we think of as "thinking".

The **motor processor** receives an action from the cognitive processor and instructs the muscles to execute it. There's an implicit **feedback** loop here: the effect of the action (either on the position of your body or on the state of the world) can be observed by your senses, and used to correct the motion in a continuous process. Finally, there is a component corresponding to your **attention**, which might be thought of like a thread of control in a computer system.

Note that this model isn't meant to reflect the anatomy of your nervous system. There probably isn't a single area in your brain corresponding to the perceptual processor, for example. But it's a useful abstraction nevertheless.

We'll look at each of these parts in more detail, starting with the processors.

4.6 Memories



- Memory properties
  - Encoding: type of things stored
  - Size: number of things stored
  - Decay time: how long memory lasts

Each component of our model has some properties. For example, memories are characterized by three properties: encoding, size, and decay time.

4.7 Short-Term Sensory Store

- Visual information store
  - encoded as physical image
  - size ~ 17 [7-17] letters
  - decay ~ 200 ms [70-1000 ms]
- Auditory information store
  - encoded as physical sound
  - size ~ 5 [4.4-6.2] letters
  - decay ~ 1500 ms [900-3500 ms]

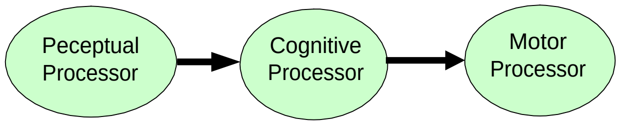
The **visual image store** is basically an image frame from the eyes. It isn't encoded as pixels, but as physical features of the image, such as curvature, length, edges. It retains physical features like intensity that may be discarded in higher-level memories (like the working memory). We measure its size in letters because psych studies have used letters as a convenient stimulus for measuring the properties of the VIS; this doesn't mean that letters are represented symbolically in the VIS. The VIS memory is fleeting, decaying in a few hundred milliseconds.

The **auditory image store** is a buffer for physical sound. Its size is much smaller than the VIS (in terms of letters), but lasts longer – seconds, rather than tenths of a second.

Both of these stores are **preattentional**; that is, they don't need the spotlight of attention to focus on them in order to be collected and stored. Attention can be focused on the visual or auditory stimulus after the fact. That accounts for phenomena like "What did you say? Oh yeah."

4.8 Processors

- Processors have a cycle time
  - Tp ~ 100ms [50-200 ms]
  - Tc ~ 70ms [30-100 ms]
  - Tm ~ 70ms [25-170 ms]



- Fastman may be 10x faster than Slowman

Turning to the processors, the main property of a processor is its **cycle time**, which is analogous to the cycle time of a computer processor. It's the time needed to accept one input and produce one output.

Like all parameters in the MHP, the cycle times shown above are derived from a survey of psychological studies. Each parameter is specified with a typical value and a range of reported values. For example, the typical cycle time for perceptual processor, Tp, is 100 milliseconds, but studies have reported between 50 and 200 milliseconds. The reason for the range is not only variance in individual humans; it is also varies with conditions. For example, the perceptual processor is faster (shorter cycle time) for more intense stimuli, and slower for weak stimuli. You can't read as fast in the dark. Similarly, your cognitive processor actually works faster under load! Consider how fast your mind works when you're driving or playing a video game, relative to sitting quietly and reading. The cognitive processor is also faster on practiced tasks.

It's reasonable, when we're making engineering decisions, to deal with this uncertainty by using all three numbers, not only the nominal value but also the range. Card, Moran, & Newell gave names to these three imaginary humans: Fastman, whose times are all as fast as possible; Slowman, whose times are all slow; and Middleman, whose times are all typical.

4.9 Perceptual Fusion

- Two stimuli within the same PP cycle (Tp ~ 100ms) appear fused
- Consequences
  - 1/ Tp frames/sec is enough to perceive a moving picture (10 fps OK, 20 fps smooth)
  - Computer response < Tp feels instantaneous
  - Causality is strongly influenced by fusion

One interesting effect of the perceptual processor is **perceptual fusion**. Here's an intuition for how fusion works. Every cycle, the perceptual processor grabs a frame (snaps a picture). 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.

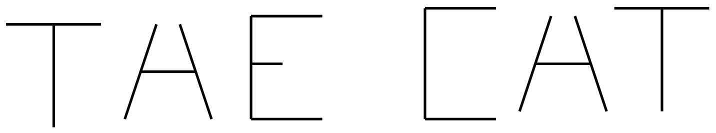
Perceptual fusion is responsible for the way we perceive a sequence of movie frames as a moving picture, so the parameters of the perceptual processor give us a lower bound on the frame rate for believable animation. 10 frames per second is good for Middleman, but 20 frames per second is better for Fastman (remember that Fastman's T_p = 50 ms represents not just the quickest humans, but also the most favorable 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.

4.10 Bottom-up vs. Top-Down Perception

- Bottom-up uses features of stimulus
- Top-down uses context
  - temporal, spatial
  - draws on long-term memory



Perception is not an isolated process. It uses both **bottom-up** processing, in which the features of a stimulus are combined to identify it, and **top-down** processing, where the context of the stimulus contributes to its recognition. In visual perception, the context is usually *spatial*, i.e., what's around the stimulus. In auditory perception, the context is *temporal* -- what you heard before or after the stimulus.

Look at the two words drawn above. The middle letter of each word is exactly identical – halfway between H and A – but the effect of the context strongly influences how we see each letter.

We'll see more about this perceptual effect in a future lecture when we talk about Gestalt principles in graphic design.

4.11 Chunking

- "Chunk": unit of perception or memory
- Chunking depends on presentation and what you already know

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

  MWR CAA OLI BMF BIB

  BMW RCA AOL IBM FBI
- 3-4 digit chunking is ideal for encoding unrelated digits

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.

4.12 Attention and Perception

- Spotlight metaphor
  - Spotlight moves serially from one input channel to another
  - Visual dominance: easier to attend to visual channels than auditory channels
  - All stimuli within spotlighted channel are processed in parallel
- Whether you want to or not

Let's look at how attention is involved with perception. The metaphor used by cognitive psychologists for how attention behaves in perception is the **spotlight**: you can focus your attention (and your perceptual processor) on only one input channel in your environment at a time. This input channel might be a location in your visual field, or it might be a location or voice in your auditory field. Humans are very visually-oriented, so it turns out to be easier to attend to visual channels than auditory channels.

Once you've focused your attention on a particular channel, all the stimuli within the area of the "spotlight" are then processed, whether you mean to or not. This can cause **interference**, as the next demonstration shows.

4.13 Say the Colors of These Words Aloud

Book
Pencil
Slide
Window
Car
Hat

Here's a little demonstration of interference. Say the **colors** of each of these words aloud, and time yourself.

4.14 Now Do It Again

Green
Orange
Red
Black
Pink
Blue

Now do it again – say the **colors** of each word aloud. It's harder, and most people do it much slower than the previous task. Why? Because the word, which names a different color, interferes with the color we're trying to say. This is called the Stroop effect.

The lesson we should take away here is that we should choose the secondary characteristics of our displays – like the multiple dimensions of stimulus, or the context around the stimulus – to **reinforce** the message of the display, not **interfere** with it.

4.15 Cognitive Processing

- Cognitive processor
  - compares stimuli
  - selects a response
- Types of decision making
  - Skill-based
  - Rule-based
  - Knowledge-based

Let's say a little about the cognitive processor, which is responsible for making comparisons and decisions.

Cognition is a rich, complex process. The best-understood aspect of it is **skill-based** decision making. A skill is a procedure that has been learned thoroughly from practice; walking, talking, pointing, reading, driving, typing are skills most of us have learned well. Skill-based decisions are automatic responses that require little or no attention. Since skill- based decisions are very mechanical, they are easiest to describe in a mechanical model like the one we're discussing.

Two other kinds of decision making are **rule-based**, in which the human is consciously processing a set of rules of the form *if X, then do Y*; and **knowledge-based**, which involves much higher-level thinking and problem-solving. Rule-based decisions are typically made by novices at a task: when a student driver approaches an intersection, for example, they must think explicitly about what they need to do in response to each possible condition. Knowledge-based decision making is used to handle unfamiliar or unexpected problems, such as figuring out why your car won't start.

We'll focus on skill-based decision making for the purposes of this lecture, because it's well understood.

4.16 Hick-Hyman Law of Choice Reaction Time

- Reaction time depends on information content of stimulus

$$RT = c + d \log_2 1/Pr(\text{stimulus})$$

  - e.g., for N equiprobable stimuli, each requiring a different response:

$$RT = c + d \log_2 N$$

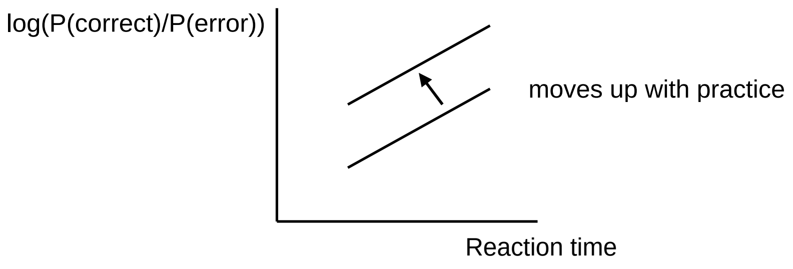
Simple reaction time – responding to a single stimulus with a single response – takes just one cycle of the human information processor, i.e. Tp+Tc+Tm.

But if the user must make a **choice** – choosing a different response for each stimulus – then the cognitive processor may have to do more work. The Hick-Hyman Law of Reaction Time shows that the number of cycles required by the cognitive processor is proportional to amount of **information** in the stimulus. For example, if there are N equally probable stimuli, each requiring a different response, then the cognitive processor needs log N cycles to decide which stimulus was actually seen and respond appropriately. So if you double the number of possible stimuli, a human's reaction time only increases by a constant.

Keep in mind that this law applies only to *skill-based* decision making; we assume that the user has practiced responding to the stimuli, and formed an internal model of the expected probability of the stimuli.

4.17 Speed-Accuracy Tradeoff

- Accuracy varies with reaction time
  - Can choose any point on curve
  - Can move curve with practice



Another important phenomenon of the cognitive processor is the fact that we can tune its 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 up 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.

4.18 Divided Attention (Multitasking)

- Resource metaphor
  - Attention is a resource that can be divided among different tasks simultaneously
- Multitasking performance depends on:
  - Task structure
    - Modality: visual vs. auditory
    - Encoding: spatial vs. verbal
    - Component: perceptual/cognitive vs. motor vs. WM
  - Difficulty
    - Easy or well-practiced tasks are easier to share

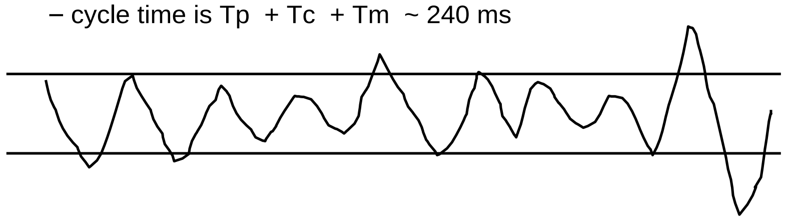
Earlier we saw the spotlight metaphor for attention. Now we'll refine it to account for our ability to handle multiple things at the same time. The **resource metaphor** regards attention as a limited resource that can be subdivided, under the human's control, among different tasks simultaneously.

Our ability to divide our attention among multiple tasks appears to depend on two things. First is the **structure** of the tasks that are trying to share our attention. Tasks with different characteristics are easier to share; tasks with similar characteristics tend to interfere. Important dimensions for task interference seem to be the **modality** of the task's input (visual or auditory), its **encoding** (e.g., spatial/graphical/sound encoding, vs. words), and the mental **components** required to perform it. For example, reading two things at the same time is much harder than reading and listening, because reading and listening use two different modalities.

The second key influence on multitasking performance is the difficulty of the task. Carrying on a conversation while driving a car is fairly effortless as long as the road is familiar and free of obstacles; when the driver must deal with traffic or navigation, conversation tends to slow down or even stop.

4.19 Motor Processing

- Open-loop control
  - Motor processor runs a program by itself
  - cycle time is Tm ~ 70 ms
- Closed-loop control
  - Muscle movements (or their effect on the world) are perceived and compared with desired result
  - cycle time is Tp + Tc + Tm ~ 240 ms



The motor processor can operate in two ways. It can run autonomously, repeatedly issuing the same instructions to the muscles. This is "open-loop" control; the motor processor receives no feedback from the perceptual system about whether its instructions are correct. With open loop control, the maximum rate of operation is just T_m.

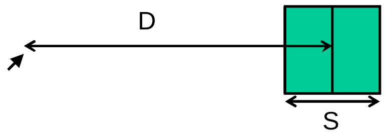
The other way is "closed-loop" control, which has a complete feedback loop. The perceptual system looks at what the motor processor did, and the cognitive system makes a decision about how to correct the movement, and then the motor system issues a new instruction. At best, the feedback loop needs one cycle of each processor to run, or $T_p + T_c + T_m$ ~ 240 ms.

Here's a simple but interesting experiment that you can try: take a sheet of lined paper and scribble a sawtooth wave back and forth between two lines, going as fast as you can but trying to hit the lines exactly on every peak and trough. Do it for 5 seconds. The frequency of the sawtooth carrier wave is dictated by open-loop control, so you can use it to derive your T_m. The frequency of the wave's **envelope**, the corrections you had to make to get your scribble back to the lines, is closed-loop control. You can use that to derive your value of $T_p + T_c$.

4.20 Fitts's Law

- Fitt's Law
  - Time T to move your hand to a target of size S at distance D away is:

$$T = RT + MT = a + b \log (2D/S)$$

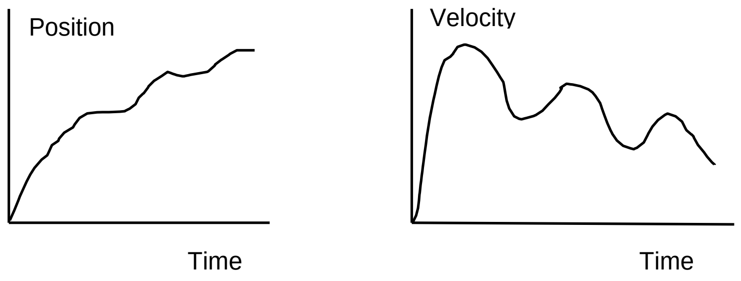


  - Depends only on index of difficulty log(2D/S)

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.

4.21 Explanation of Fitts's Law

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



We can explain Fitts's Law by appealing to the human information processing model. Fitt's Law relies on closed-loop control. 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 $\varepsilon^2 D$. This process repeats, with the error decreasing geometrically, until *n* iterations have brought your hand within the target – i.e., $\varepsilon^n D \le ½ S$. Solving for *n*, and letting the total time $T = n (T_p + T_c + T_m)$, we get:

$$T = a + b \log (2D/S)$$

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

4.22 Implications of Fitts's Law

- Targets at screen edge are easy to hit
  - Mac menubar beats Windows menubar
  - Unclickable margins are foolish
- Hierarchical menus are hard to hit
  - Gimp/GTK: instantly closes menu
  - Windows: .5 s timeout destroys causality
  - Mac does it right: triangular zone
- Linear popup menus vs. pie menus

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 half the size S of the target, so T = a + b log (2D/S) solves to the minimum time T=a+b.) 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.
- As we discussed last week, hierarchical submenus are hard to use, because of the correction cycles the user is forced to spend getting the mouse pointer carefully over into the submenu. 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 Slowman), so it destroys perceptual fusion and our sense of causality. Intentionally moving the mousedown to the next menu results in a noticeable delay. The Mac gets a Hall of Fame nod here, for doing it right with a triangular zone of activation for 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 are often invisible!
- 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).

4.23 Power Law of Practice

- Time Tn to do a task the nth time is:

$$T_n = T_1 n^{-\alpha}$$

α is typically 0.2-0.6

An important feature of the entire perceptual-cognitive-motor system is that the time to do a task decreases with practice. In particular, the time decreases according to the power law, shown above. The power law describes a linear curve on a log-log scale of time and number of trials.

In practice, the power law means that novices get rapidly better at a task with practice, but then their performance levels off to nearly flat (although still slowly improving).

4.24 Working Memory (WM)

- Small capacity: 7 ± 2 "chunks"
- Fast decay (7 [5-226] sec)
- **Maintenance rehearsal** fends off decay
- Interference causes faster decay

Working memory is where you do your conscious thinking. Working memory is where the cognitive processor gets its operands and drops its results. 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, due to George Miller (unrelated), is that the capacity of working memory is roughly 7 ± 2 chunks.

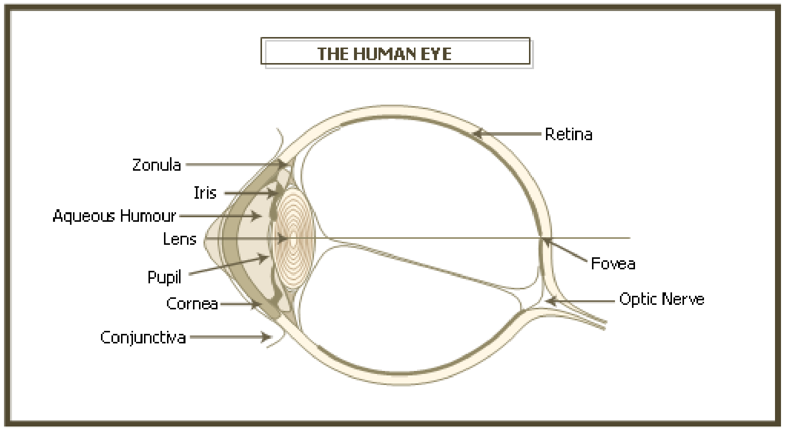
Working memory decays in tens of seconds. **Maintenance rehearsal** – repeating the items to yourself – fends off this decay, much as DRAM must refresh itself. Maintenance rehearsal requires attentional resources. But distraction destroys working memory. A particularly strong kind of distraction is **interference** – stimuli that activate several conflicting chunks are much harder to retain. Recall the Stroop effect from earlier in this lecture: not only does it slow down perception, but it also inhibits our ability to retain the colors in working memory.

4.25 Long-term Memory (LTM)

- Huge capacity
- Little decay
- **Elaborative rehearsal** moves chunks from WM to LTM by making connections with other chunks

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 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. Elaborative rehearsal requires attention resources as well.

4.26 The Eye



OK, we’ve looked at perception, cognition, motor skills, and memory. We’ll conclude our discussion of the human machine today by considering the vision system in a little more detail, since vision is the primary way that a graphical user interface communicates to the user.

Here are key parts of the anatomy of the eye:

- The **cornea** is the transparent, curved membrane on the front of the eye.
- The **aqueous humor** fills the cavity between the cornea and the lens, and provides most of the optical power of the eye because of the large difference between its refractive index and the refractive index of the air outside the cornea.
- The **iris** is the colored part of the eye, which covers the lens. It is an opaque muscle, with a hole in the center called the **pupil** that lets light through to fall on the lens. The iris opens and closes the

pupil depending on the intensity of light; it opens in dim light, and closes in bright light.

- The **lens** focuses light. Under muscle control, it can move forward and backward, and also get thinner or fatter to change its focal length.
- The **retina** is the surface of the inside of the eye, which is covered with light-sensitive receptor cells.
- The **fovea** is the spot where the optical axis (center of the lens) impinges on the retina. The highest density of photoreceptors can be found in the fovea; the fovea is the center of your visual field.

4.27 Photoreceptors

- Rods
  - Only one kind (peak response in green wavelengths)
  - Sensitive to low light ("scotopic vision")
    - Multiple nearby rods aggregated into a single nerve signal
  - Saturated at moderate light intensity ("photopic vision")
    - Cones do most of the vision under photopic conditions
- Cones
  - Operate in brighter light
  - Three kinds: S(hort), M(edium), L(ong)
  - S cones are very weak, centered in blue wavelengths
  - M and L cones are more powerful, overlapping
  - M centered in green, L in yellow (but called "red")

There are two kinds of photoreceptor cells in the retina. **Rods** operate under low-light conditions – night vision. There is only one kind of rod, with one frequency response curve centered in green wavelengths, so rods don't provide color vision. Rods saturate at moderate intensities of light, so they contribute little to daytime vision. **Cones** respond only in brighter light. There are three kinds of cones, called S, M, and L after the centers of their wavelength peaks. S cones have very weak frequency response centered in blue. M and L cones are two orders of magnitude stronger, and their frequency response curves nearly overlap.

4.28 Signals from Photoreceptors

- Brightness
  M + L + rods
- Red-green difference
  L - M
- Blue-yellow difference
  weighted sum of S, M, L

The rods and cones do not send their signals directly to the visual cortex; instead, the signals are recombined into three channels. One channel is **brightness**, produced by the M and L cones and the rods. This is the only channel really active at night. The other two channels convey color **differences**. High responses mean red, and low responses indicate green. These difference channels drive the theory of **opponent colors**: red and green are good contrasting colors because they drive the red-green channel to opposite extremes. Similarly, black/white and blue/yellow are good contrasting pairs.

4.29 Color Blindness

- Red-green color blindness (protonopia & deuteranopia)
  - 8% of males
  - 0.4% of females
- Blue-yellow color blindness (tritanopia)
  - Far more rare
- Guideline: don't depend solely on color distinctions
  - use redundant signals: brightness, location, shape

Color deficiency (“color blindness”) affects a significant fraction of human beings. An overwhelming number of them are male.

There are three kinds of color deficiency, which we can understand better now that we understand a little about the eye’s anatomy:

- **Protonopia** is missing or bad L cones. The consequence is reduced sensitivity to red-green differences (the L-M channel is weaker), and reds are perceived as darker than normal.
- **Deuteranopia** is caused by missing or malfunctioning M cones. Red-green difference sensitivity is reduced, but reds do not appear darker.
- **Tritanopia** is caused by missing or malfunctioning S cones, and results in blue-yellow insensitivity.

Since color blindness affects so many people, it is essential to take it into account when you are deciding how to use color in a user interface. Don’t depend solely on color distinctions, particularly red-green distinctions, for conveying information. Microsoft Office applications fail in this respect: red wavy underlines indicate spelling errors, while identical green wavy underlines indicate grammar errors.

Traffic lights are another source of problems. How do red-green color-blind people know whether the light is green or red? Fortunately, there’s a spatial cue: red is always above (or to the right of) green. Protonopia sufferers (as opposed to deuteranopians) have an additional advantage: the red light looks darker than the green light.

4.30 Chromatic Aberration

- Different wavelengths focus differently
  - Highly separated wavelengths (red & blue) can′t be focused simultaneously
- Guideline: don′t use red-on-blue text
  - It looks fuzzy and hurts to read

The refractive index of the lens varies with the wavelength of the light passing through it; just like a prism, different wavelengths are bent at different angles. So your eye needs to focus differently on red features than it does on blue features.

As a result, an edge between widely-separated wavelengths – like blue and red – simply can’t be focused. It always looks a little fuzzy. So blue-on-red or red-on-blue text is painful to read, and should be avoided at all costs.

Apple’s ForceQuit tool in Mac OS X, which allows users to shut down misbehaving applications, unfortunately falls into this trap. In its dialog, unresponding applications are helpfully displayed in red. But the selection is a blue highlight. The result is incredibly hard to read.

4.31 Blue Details Are Hard to Resolve

- Fovea has no S cones
  - Can′t resolve small blue features (unless they have high contrast with background)
- Lens and aqueous humor turn yellow with age
  - Blue wavelengths are filtered out
- Lens weakens with age
  - Blue is harder to focus
- Guideline: don′t use blue against dark backgrounds where small details matter (text!)

A number of anatomical details conspire to make blue a bad color choice when small details matter.

First, the fovea has very few S cones, so you can’t easily see blue features in the center of your vision (unless they have high contrast with the background, activating the M and L cones).

Second, older eyes are far less sensitive to blue, because the lens and aqueous humor slowly grow yellower, filtering out the blue wavelengths.

Finally, the lens gets weaker with age. Blue is at one extreme of its focusing range, so older eyes can’t focus blue features as well.

As a result, avoid blue text, particularly small blue text.

4.32 Fovea Has No Rods

- Rods are more sensitive to dim light
- In scotopic conditions, peripheral vision (rod-rich) is better than foveal vision
  - Easier to see a dim star if you don't look directly at it

Incidentally, the fovea has no rods, either. That explains why it's easier to see a dim star if you don't look beside it, rather than directly at it.

Lecture 5: Output Models

5.1 Today's Topics

- Output modes
- Automatic redraw
- Antialiasing & subpixel rendering
- Color models

Today's lecture continues our look into the mechanics of implementing user interfaces, by looking at **output** in more detail.

Our 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. Presumably you've already encountered at least one GUI toolkit, probably Java Swing. 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.

5.2 Three Output Models

- Components
  - Graphical objects arranged in a tree with automatic redraw
  - Example: Lable object, Line object
  - Also called: views, interactors, widgets, contrlo s, retained graphics
- Strokes
  - High-level drawing primitives: lines, shapes, curves, text
  - Example: drawText() method, drawLine(e) method
  - Also called: vector graphics, structured graphics
- Pixels
  - 2D array of pixels
  - Also called: raster, image, bitmap

There are basically three ways to represent the output of a graphical user interface.

Components is the same as the view hierarchy we discussed last week. 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.

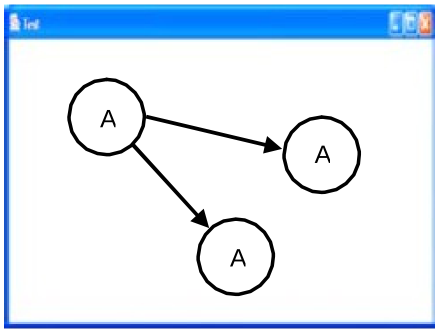
Strokes draws output by making calls to high-level drawing primitives, like drawLine, drawRectangle, drawArc, and drawText.

Pixels regards the screen as an array of pixels and deals with the pixels directly.

All three output models appear in virtually every modern GUI application. The component model always appears at the very top level, for windows, and often for components 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, a component 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 model do each of the following representations use? HTML (component); Postscript laser printer (stroke input, pixel output); plotter (stroke input and output); PDF (stroke); LCD panel (pixel).

5.3 Example: Designing a Graph View



Since every application uses all three models, the design question becomes: at which points in your application do you want to step down into a lower-level output model? Here's an example. Suppose you want to build a view that displays a graph of nodes and edges.

One approach would represent each node and edge in the graph by a component. Each node in turn might have two components, a rectangle and a label. Eventually, you'll get down to primitive components available in your GUI toolkit. Most GUI toolkits provide a label component; most don't provide a primitive circle component. One notable exception is Amulet, which has component equivalents for all the common drawing primitives. This would be a **pure component model**, at least from your application's point of view – stroke output and pixel output would still happen, but inside primitive components 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: drawRectangle for the node outlines, drawText for the labels, drawLine for the edges. This would be a **pure stroke model**.

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 model** 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 models for the graph view are certainly possible, in which some parts of the output use one model, and others use another model. The graph view might use components for nodes, but draw the edges itself as strokes. It might draw all the lines itself, but use label components for the text.

5.4 Issues in Choosing Output Models

- Layout
- Input
- Redraw
- Drawing order
- Heavyweight objects
- Device dependence

Layout: Components remember where they were put, and draw themselves there. They also support automatic layout. With stroke or pixel models, you have to figure out (at drawing time) where each piece goes, and put it there.

Input: Components 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 components, but not for strokes. If a graph node is a component, 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 components don't bother, simply redrawing everything whenever some part of the view needs to be redrawn.

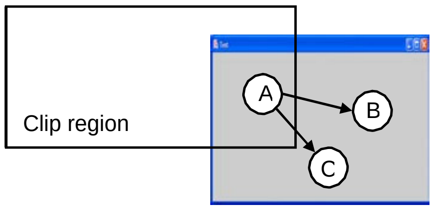
Drawing order: It's easy for a parent to draw before (underneath) or after (on top of) all of its children. But it's not easy to interleave parent drawing with child drawing. So if you're using a hybrid model, with some parts of your view represented as components and others as strokes, then the components and strokes generally fall in two separate layers, and you can't have any complicated z-ordering relationships between strokes and components.

Heavyweight objects: Every component must be an object (and even an object with no fields costs about 20 bytes in Java). As we've seen, the view hierarchy is overloaded not just with drawing functions but also with event dispatch, automatic redraw, and automatic layout, so that further bulks up the class. The flyweight pattern used by InterView's Glyphs can reduce this cost somewhat. But views derived from large amounts of data – say, a 100,000-node graph – generally can't use a component model.

Device dependence: The stroke model 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 model, 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.

5.5 Drawing in the Component Model

- 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 = intersection

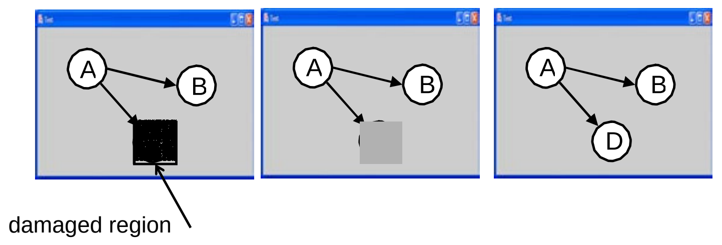


Here's how drawing works in the component model. Drawing is a top-down process: starting from the root of the component tree, each component draws itself, then draws each of its children recursively. The process is optimized by passing a **clipping region** to each component, indicating the area of the 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 a component 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 component, the clipping region might be the entire screen. As drawing descends the component tree, however, the clipping region is intersected with each component's bounding box. So the clipping region for a component deep in the tree is the intersection of the bounding boxes of its ancestors.

For high performance, the clipping region is normally rectangular, using component **bounding boxes** rather than the components' 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.

5.6 Damage and Automatic Redraw

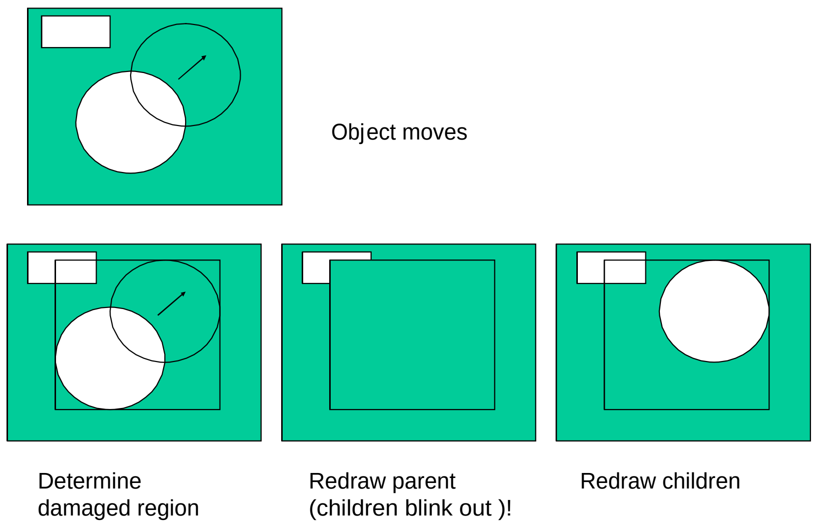


When a component needs to change its appearance, it doesn't repaint itself directly. It *can't*, because the drawing process has to occur top-down through the component hierarchy: the component's ancestors and older siblings need to have a chance to paint themselves underneath it. (So, in Java, even though a component can call its paint() method directly, you shouldn't do it!)

Instead, the component 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 component; but complex components 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 components 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 components. 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 component tree from the root.

5.7 Naïve Redraw Causes Flashing Effects

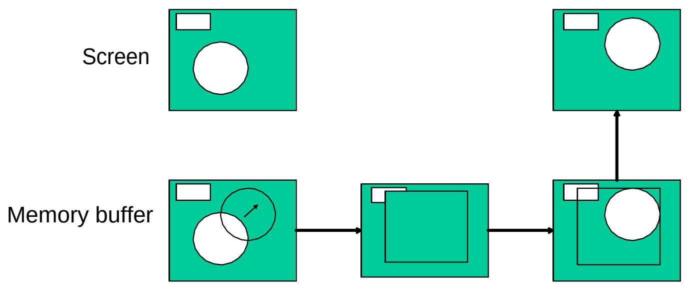


There's an unfortunate side-effect of the automatic damage/redraw algorithm. If we draw a component tree directly to the screen, then moving a component can make the screen appear to flash – objects flickering while they move, and nearby objects flickering as well.

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. 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.

5.8 Double-Bufering

- Double-buffering solves the flashing problem



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.

5.9 Stroke Model

- Drawing surface
  - Also called drawalb e (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

We've already considered the component model in some detail. So now, let's look at the stroke model.

Every stroke model has some notion of a drawing surface. The screen is only one 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.

5.10 Converting Strokes to Pixels

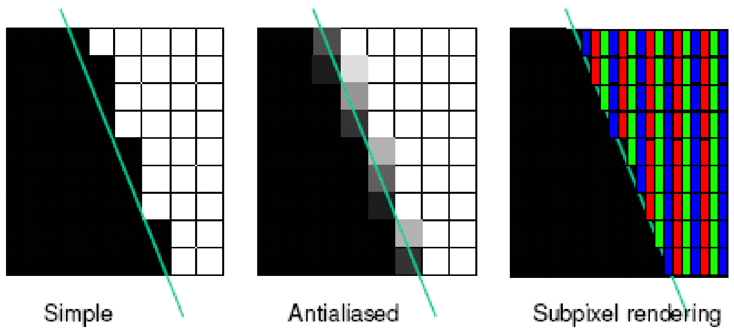
- Is (0,0) the center of the top-left pixel, or is it the upper left corner of the pixel?
  - MS Win: center of pixel
  - Java: upper left corner
- Where is line (0,0) - (10,0) actuay drawn?
  - MS Win: endpoint pixel excluded
  - Java Graphics: pen hangs down and right
  - Java Graphcs2D: antialiased pen, optional ½ pixel adjustments made for compatibility.
- Where is empty rectangle (0,0) – (10,10) drawn?
  - MS Win: connectng those pixel
  - Java: extends one row below and one coumn right
- Where is filled rectangle (0,0) – (10,10) drawn?
  - MS Win: 121 pixels
  - Java: 100 pixels

When you're using a stroke model, it's important to understand how the strokes are actually converted into pixels. Different platforms make different choices.

One question concerns how stroke coordinates, which represent zero-dimensional points, are translated into pixel coordinates, which are 2-dimensional squares. Microsoft Windows places the stroke coordinate at the center of the corresponding pixel; Java's stroke model places the stroke coordinates between pixels.

The other questions concern which pixels are actually drawn when you request a line or a rectangle.

5.11 Antialiasing and Subpixel Rendering



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

5.12 Color Models

- RGB: cube
  - Red, green, blue
- HSV: hexagonal cone
  - Hue: kind of color
    - Angle around cone
  - Saturation: amount of pure color
    - 0% = gray, 100% = pure color
  - Value: brightness
    - 0% = dark, 100% = bright
- HLS: double-hexagonal cone
  - Hue, lightness, saturation
  - Pulls up center of HSV model, so that only white has lightness 1.0 and pure colors have lightness 0.5
- Cyan-Magenta-Yellow(-Black)
  - Used for printing, where pigments absorb wavelengths instead of generating them.

We learned a bit about how humans perceive color when we talked about human capabilities. Now let's look at how colors are represented in GUI software.

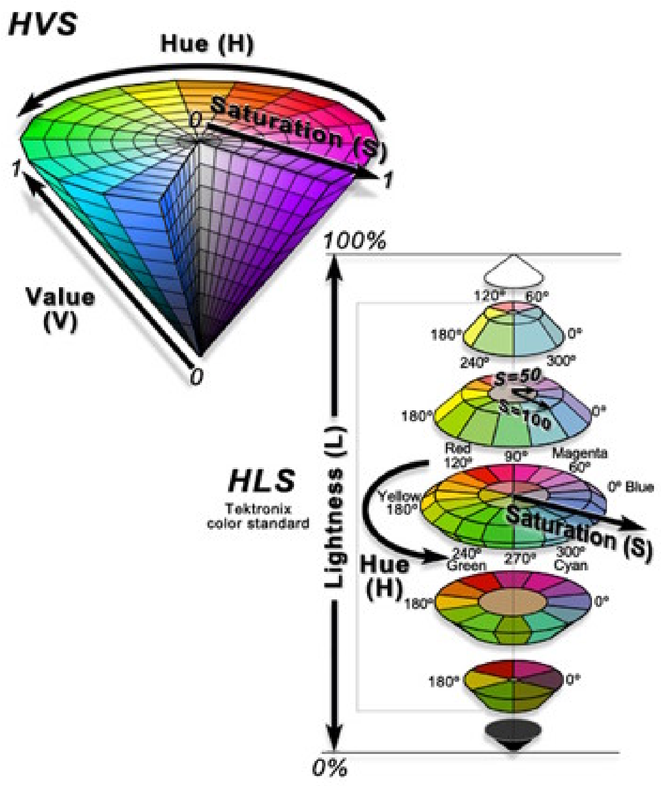
At the lowest level, the RGB model rules. The RGB model is a unit cube, with (0,0,0) corresponding to black, (1, 1, 1) corresponding to white, and the three dimensions measuring levels of red, green, and blue. The RGB model is used directly by CRT and LCD monitors for display, since each pixel in a monitor has separate red, green, and blue components.

HSV (hue, saturation value) is a better model for how humans perceive color, and more useful for building usable interfaces. HSV is a cone. We've already encountered hue and value in our discussion of visual variables. Saturation is the degree of color, as opposed to grayness. Colors with zero saturation are shades of gray; colors with 100% saturation are pure colors.

HLS (hue, lightness, saturation) is a symmetrical relative of the HSV model, which is elegant. See the pictures on the next page.

Finally, the CMYK (cyan, magenta, yellow, and sometimes black) is similar to the RGB model, but used for print colors.

5.13 HSV & HLS



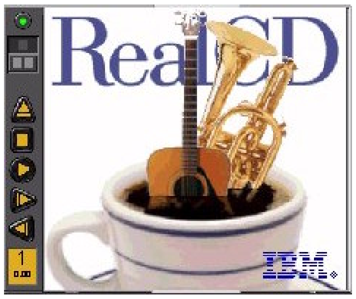
5.14 Transparency

- 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

Modern color models add a fourth channel: the pixel's **alpha** value, which is its transparency. Simple image formats like GIF support only two values of alpha: 1 (opaque) or 0 (transparent). PNG has better support, allowing image pixels to be translucent, with alpha values between 0 and 1.

Lecture 6: Models & Metaphors

6.1 UI Hall of Fame or Shame?



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

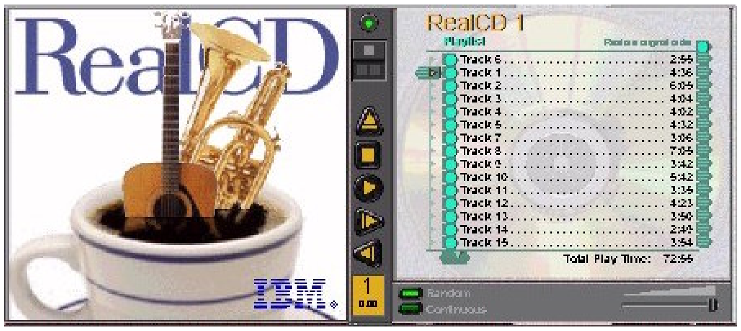
Why is it called "Real"? Because its designers based it on a real-world object: a plastic CD case. This interface has a metaphor, an analogue in the real world. Metaphors are one way to make an interface "intuitive," since users can make guesses about how it will work based on what they already know about the interface's metaphor. Unfortunately, the designers' careful adherence to this metaphor produced some remarkable effects, none of them good. Here's how RealCD looks when it first starts up. Notice that the UI is dominated by artwork, just like the outside of a CD case is dominated by the cover art. That big RealCD logo is just that – static artwork. Clicking on it does nothing.

There's an obvious problem with the choice of metaphor, of course: a CD case doesn't actually play CDs. The designers had to find a place for the player controls – which, remember, serve the primary task of the interface – so they arrayed them vertically along the case hinge. The metaphor is dictating control layout, against all other considerations.

Slavish adherence to the metaphor also drove the designers to disregard all consistency with other desktop applications. Where is this window's close box? How do I shut it down?

You might be able to guess, but is it "intuitive?" Learnability comes from more than just metaphor.

6.2 UI Hall of Shame!



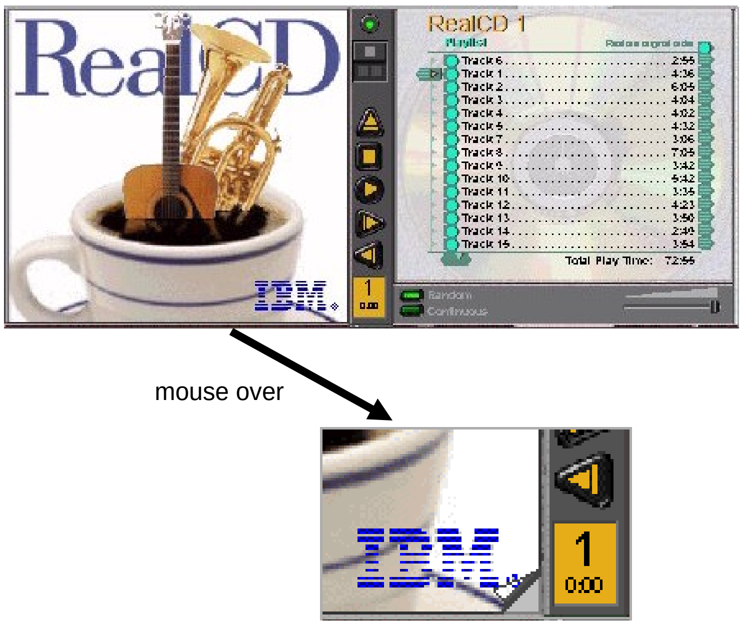
```
Track 6 ......................... 2:55
Track 1 ......................... 4:36
Track 2 ......................... 6:05
Track 3 ......................... 4:04
Track 4 ......................... 4:02
Track 5 ......................... 4:32
Track 7 ......................... 3:06
Track 8 ......................... 7:05
Track 9 ......................... 3:42
Track 10 ........................ 5:42
Track 11 ........................ 3:35
Track 12 ........................ 4:23
Track 13 ........................ 3:50
Track 14 ........................ 2:49
Track 15 ........................ 3:54
                 Total Play Time: 72:55
```

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?

6.3 UI Hall of Shame



Track 6	2:55
Track 1	4:36
Track 2	6:05
Track 3	4:04
Track 4	4:02
Track 5	4:32
Track 7	3:06
Track 8	7:05
Track 9	3:42
Track 10	5:42
Track 11	3:35
Track 12	4:23
Track 13	3:50
Track 14	2:49
Track 15	3:54
Total Play Time:	72:55

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

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

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

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

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

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

6.4 Today's Topics

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

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

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

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

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

6.5 Models

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

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

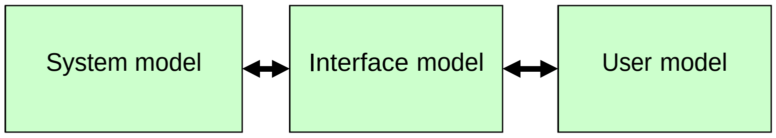
Consider image editing software. Programs like Photoshop and Gimp use a pixel editing model, in which an image is represented by an array of pixels (plus a stack of layers). Programs like Visio and Illustrator, on the other hand, use a structured graphics model, in which an image is represented by a collection of graphical objects, like lines, rectangles, circles, and text. In this case, the choice of model strongly constrains the kinds of operations available to a user. You can easily tweak individual pixels in Photoshop, but you can't easily move an object once you've drawn it into the picture.

Similarly, most modern text editors model a text file as a single string, in which line endings are just like other characters. But it doesn't have to be this way. Some editors represent a text file as a list of lines instead. When this implementation model is exposed in the user interface, as in the vi text editor, line endings can't be deleted in the same way as other characters. Vi has a special join command for deleting line endings.

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

6.6 Models in UI Design

- Three models are relevant to UI design:



The preceding discussion hinted that there are actually several models you have to worry about in UI design:

- The **system model** (sometimes called implementation model) is how the system actually works.
- The **interface model** (or manifest model) is the model that the system presents to the user.

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

6.7 Interface Model Hides System Model

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

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

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

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

6.8 User Model May Be Wrong

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

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

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

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

The problem here is that the thermostat isn't effectively communicating its model to the user. In particular, there isn't enough feedback about what the furnace is doing for the user to form the right model.

6.9 Interaction Styles

- Command language
- Menus & forms
- Direct manipulation

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

6.10 Command Language

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

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

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

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

6.11 Menus and Forms

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

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

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

6.12 Direct Manipulation

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

    - Files and folders on a desktop
    - Scrollbar
    - Dragging to resize a rectangle
    - Selecting text
- Visual representation and physical interaction are important

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

- 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.

- 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.

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

Why is direct manipulation so powerful? It exploits perceptual and motor skills of the human machine – and depends less on linguistic skills than command or menu/form interfaces.

6.13 Comparison of Interaction Styles

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

Error messages: DM rarely needs them. No error message when you drag a scrollbar "too far", for example. **Knowledge in the head vs. the world.** Command languages require the user to put a lot of knowledge into their heads, by training, practice, etc. (Or else compensate by having manuals, reference cards, or online help close at hand while using the system.) Menus and forms put much more information into the world. Well- designed DM also has information in the world, from the affordances and constraints of the visual metaphor. **Efficiency**: experts can be very efficient with command languages (no need to scan system prompts; able to reuse commands in scripts and history). Efficient performance with menus and form interfaces demands good shortcuts (e.g. keyboard shortcuts, tabbing between form fields, typeahead). Efficient performance with DMs is possible when the DM is appropriate to the task (and depends on shortcuts and Fitts's Law); but using a DM for a task it isn't designed for may turn into manual labor.

User experience: command languages best for expert users – frequent, well-trained. Menus/forms and DMs better for novices and infrequent users.

Synchrony: Command languages are synchronous (first the user types a complete command, then the system does it). So are menu systems and forms; e.g. web model. DM is asynchronous: user can point mouse anywhere and do anything at any time.

Programming difficulty: Command languages are relatively easy: parsing text, rigid requirements.

Menus and forms have substantial toolkit support; e.g., the web browser and HTML, Java Swing components. DM is hardest to program: have to draw, handle low-level input (keyboard, mouse), display feedback. Very few off- the-shelf components available to help. DM can't be done easily on the Web without Java or Flash. **Accessibility**: command and menu/form interfaces are more textual, so easier for vision-impaired users to read with screen readers. DM interfaces are much harder for these users.

6.14 Direct Manipulation Cues

- Affordances
- Constraints
- Natural mapping
- Visibility
- Feedback

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

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

6.15 Affordances

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

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

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

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

6.16 Natural Mapping

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

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

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

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

Other good examples of mapping include:

- Stove burners. Many stoves have four burners arranged in a square, and four control knobs arranged in a row. Which knobs control which burners? Most stoves don't make any attempt to provide a natural mapping.
- Car turn signals. The turn signal switch in most cars is a stalk that moves up and down, but the function it controls is a signal for left or right turn. So the mapping is not direct, but it is nevertheless natural. Why?
- An audio mixer for DJs (proposed by Max Van Kleek for the Hall of Fame) has two sets of identical controls, one for each turntable being mixed. The mixer is designed to sit in between the turntables, so that the left controls affect the turntable to the left of the mixer, and the right controls affect the turntable to the right. The mapping here is direct.

6.17 Visibility

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

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

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

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

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

6.18 Feedback

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

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

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

6.19 Modeling Human Error

- Description error
- Capture error
- Mode error

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

6.20 Description Error

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

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

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

6.21 Capture Error

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

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

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

6.22 Mode Error

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

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

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

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

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

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

6.23 Metaphors

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

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

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

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

6.24 Dangers of Metaphors

- Hard to find
- Deceptive
- Constraining
- Breaking the metaphor
- Use of a metaphor doesn't excuse bad communication of the model:
  - RealCD's bad affordances, visibility

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

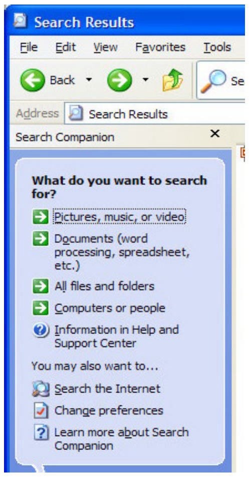
Metaphors can also be deceptive, leading users to infer behavior that your interface doesn’t provide. Sure, it looks like a book, but can I write in the margin? Can I rip out a page? Metaphors can also be constraining. Strict adherence to the desktop metaphor wouldn’t scale, because documents would always be full-size like they are in the real world.

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

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

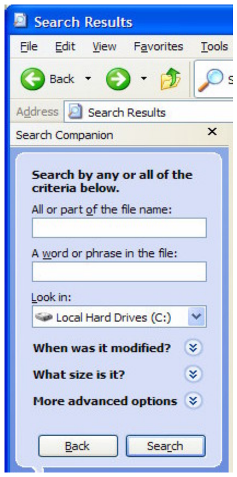
Lecture 7: Input Models

7.1 UI Hall of Fame or Hall of Shame?


Search Results
File Edit View Favorites Tools
Back
Address Search Results
Search Companion
Pictures, music, or video
Documents (word processing, spreadsheet, etc.)
All files and folders
Computers or people
Information in Help and Support Center
Search the Internet
Change preferences
Learn more about Search Companion

What do you want to search for?

You may also want to...



Search by any or all of the criteria below.

All or part of the file name:

A word or phrase in the file:

Look in: Local Hard Drives (C:)

This is the Windows XP Search Companion. It appears when you press the Search button on a Windows Explorer toolbar, and is primarily intended for finding files on your hard disk.

An interesting feature of this interface is that, rather than giving a textbox for search keywords right away, it first asks you to specify what kind of file you're looking for. There's some logic to this design decision, because it turns out that different search criteria are appropriate for different kinds of files. For example, if you select "Picture, music, and video", the next step of the dialog won't both asking for a word or phrase *inside* the file, since these kinds of files are not textual. Similarly, if you select "Documents", the next step of the dialog will ask not only for search keywords, but also for the approximate time since you last edited the file, since most documents are sought for editing purposes (while most media files are sought for playing purposes).

Unfortunately, to a frequent user, the demand that you specify the file's type *first* feels jarring and hard to answer. The categories are not disjoint, so the decision isn't always easy. Are HTML files and simple text files included in "Documents", or only Microsoft Office files? Some of the categories are bizarre – "computers or people"? Why is "Internet" a completely separate category, and why does Help get a different icon than the rest?

Perhaps the worst problem in the category list is that the answer that frequent users are most likely to want – "All files and folders", to be sure that the search won't miss anything – is actually buried in the middle of the list, where it's hardest to find and click.

This interface is clearly designed for novice users. Hence the **wizard** design, a fixed sequence of carefully guided steps. And hence the cute animated cartoon dog, which some people in class found condescending by its mere presence. It's still an open question whether cartoon characters like this dog and the Paperclip are more helpful or harmful to good user interface design. So far, experiments with characters in serious commercial interfaces (designed for productivity rather than entertainment) have been largely unsuccessful.

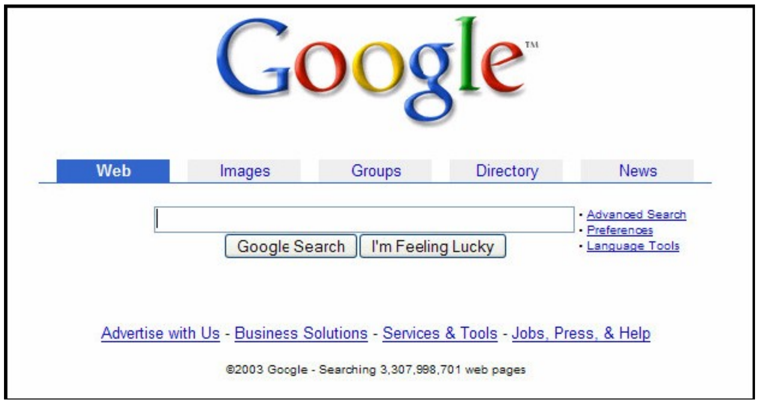
The animated dog does have one advantage: it's a very visible mode status indicator. You won't accidentally leave the Windows Explorer in search mode, because the dog will get your attention and motivate you to find a way to get rid of it -- which is not trivial, since there's no obvious Cancel button.

Another problem with this wizard is that the Back button on toolbar is easy to confuse with the Back button in the dialog. The user thinks "this isn't what I want, I'll go Back", but then reaches habitually for the Back button in the toolbar, which backs up the main Explorer window instead of the Search Companion pane. This is probably a **capture error**, because of the effect of habit, but it also has some features of a **description error.**

It turns out that "Change preferences" leads to a menu where you can turn off the dog. He doesn't disappear instantly, but turns insouciantly and trots off in a huff. The preferences menu also offers an

Advanced mode which automatically defaults to searching all files & folders.

7.2 Hall of Fame or Shame?


Google™
Web
Images
Groups
Directory
News
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help
©2003 Google - Searching 3,307,998,701 web pages

In contrast to the previous example, here's Google's start page. Google is an outstanding example of a heuristic we'll see today: **Aesthetic and minimalist design.** Its interface is as simple as possible. Unnecessary features and hyperlinks are omitted, lots of whitespace is used. Google is fast to load and trivial to use.

But maybe Google goes a little too far! Take the perspective of a completely novice user coming to Google for the first time.

- What does Google actually do? The front page doesn't say.
- What should be typed into the text box? It has no caption at all.
- The button labels are almost gibberish. "Google Search" isn't meaningful English (although it's gradually becoming more meaningful as Google enters the language as a noun, verb, and adjective). And what does "I'm Feeling Lucky" mean?
- Where is Help? Turns out it's buried at the bottom, along with "Jobs & Press".

Although these problems would be easy for Google to fix, they are actually minor, because Google's interface is simple enough that it can be learned by only a small amount of exploration. (Except perhaps for the I'm Feeling Lucky button, which probably remains a mystery until a user is curious enough to hunt for the help. After all, maybe it does a random choice from the search results!) Notice that Google does not ask you to choose your search domain first. It picks a good default, and makes it easy to change.

7.3 Class Projects

1. Spam Control
2. Firewall Visualization
3. Lecture Player
4. Timeliner IDE
5. Kerberos/AFS Ticket Manager
6. Semantic Web By Example
7. ComicKit
8. Electronic Ballots
9. Rummikub Game
10. Airport Information Kiosk
11. Air Traffic Control
12. Firewall Visualization
13. Lecture Player
14. Timeliner IDE
15. Kerberos/AFS Ticket Manager
16. Semantic Web By Example
17. ComicKit
18. Electronic Ballots
19. Rummikub Game
20. Airport Information Kiosk
21. Air Traffic Control
22. IFC Rush Manager

In case you're curious, here are the projects that your classmates are working on. You'll have several opportunities to see what everybody is doing: some in paper prototype testing in 2 weeks, others when you do heuristic evaluation of computer prototypes, and all of them in the final presentations at the end of the course.

Incidentally, the original version of this slide used bullets instead of numbers. Then I thought about one natural question that people would ask – how many projects are there? Although it's possible to answer that question from a bulleted list, it's trivial when the list is numbered. Every kind of communication you do has a user interface, whether it's a talk or a paper or a homework assignment.

The effectiveness of a communication is strongly influenced by its usability.

7.4 Today's topics

- Input

Today's lecture continues our look into the mechanics of implementing user interfaces, by looking at **input** and **output** in more detail.

Our 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. Presumably you've already encountered at least one GUI toolkit, probably Java Swing. 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.

7.5 Hints for Debugging Output

- Something you_re drawing isn′t appearing on the screen. Why not?
  - Wrong place
  - Wrong size
  - Wrong color
  - Wrong z-order

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 0 width and 0 height, it will be completely invisible no matter what it tries to draw– everything will be clipped. 0 width and 0 height is the default for all components in Swing – you have to use automatic layout or manual setting to make it a 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? Wrong z-order: is something else drawing on top?

7.6 Why Use Events for GUI Input?

- Console I/O uses blocking procedure calls

```
print (  Enter name:  )
name = readLine();
print (  Enter phone number:  )
name = readLine();
```

  - System controls the dialogue
- GUI input uses event handling instead
  - User has much more control over the dialogue (user control and freedom)
  - User can click on almost anything

Virtually all GUI toolkits use event handling for input. Why? Recall, when you first learned to program, you probably wrote user interfaces that printed a prompt and then waited for the user to enter a response. After the user gave their answer, you produced another prompt and waited for another response. Command-line interfaces (e.g. the Unix shell) and menu-driven interfaces (e.g., Pine) have interfaces that behave this way. In this user interface style, the system has complete control over the dialogue – the order in which inputs and outputs will occur.

Interactive graphical user interfaces can't be written this way – at least, not if they care about giving the user control and freedom. One of the biggest advantages of GUIs is that a user can click anywhere on the window, invoking any command that's available at the moment, interacting with any view that's visible. In a GUI, the balance of power in the dialogue swings strongly over to the user's side.

As a result, GUI programs can't be written in a synchronous, prompt-response style. A component can't simply take over the entire input channel to wait for the user to interact with it, because the user's next input may be directed to some other component on the screen instead. So GUI programs are designed to handle input asynchronously, receiving it as events.

7.7 Kinds of Input Events

- Raw input events
  - Mouse moved
  - Mouse button pressed or released Key pressed or released
- Translated input events
  - Mouse click or double-click
  - Mouse entered or exited component Keyboard focus gained or lost Character typed

There are two major categories of input events: raw and translated.

A raw event comes right from the device driver. Mouse movements, mouse button down and up, and keyboard key down and up are the raw events seen in almost every capable GUI system. A toolkit that does not provide separate events for down and up is poorly designed, and makes it difficult or impossible to implement input effects like drag-and-drop or video game controls.

For many GUI components, the raw events are too low-level, and must be translated into higher-level events. For example, a mouse button press and release is translated into a mouse click event (assuming the mouse didn't move much between press and release – if it did, these events would be translated into a drag rather than a click). Key down and up events are translated into character typed events, which take modifiers into account to produce an ASCII character rather than a keyboard key. If you hold a key down, multiple character typed events may be generated by an autorepeat mechanism. Mouse movements and clicks also translate into keyboard focus changes. When a mouse movement causes the mouse to enter or leave a component's bounding box, entry and exit events are generated, so that the component can give feedback – e.g., visually highlighting a button, or changing the mouse cursor to a text I-bar or a pointing finger.

7.8 Properties of Input Events

- Mouse position (X,Y)
- Mouse button state
- Modifier key state (Ctrl, Shift, Alt, Meta)
- Timestamp
  - Why is timestamp important?

Input events have some or all of these properties. On most systems, all events include the modifier key state, since some mouse gestures are modified by Shift, Control, and Alt. Some systems include the mouse position and button state on all events; some put it only on mouse-related events.

The timestamp indicates when the input was received, so that the system can time features like autorepeat and double-clicking. It is essential that the timestamp be a property of the event, rather than just read from the clock when the event is handled. Events are stored in a queue, and an event may languish in the queue for an uncertain interval until the application actually handles it.

7.9 Event Queue

- Events are stored in a queue
  - User input tends to be bursty
  - Queue saves application from hard real time constraints (i.e., having to finish handling each event before next one might occur)
- Mouse moves are coalesced into a single event in queue
  - If application can t keep up, then sketched lines have very few points

User input tends to be bursty – many seconds may go by while the user is thinking, followed by a flurry of events. The event queue provides a buffer between the user and the application, so that the application doesn't have to keep up with each event in a burst. Recall that perceptual fusion means that the system has 100 milliseconds in which to respond.

Edge events (button down and up events) are all kept in the queue unchanged. But multiple events that describe a continuing state – in particular, mouse movements – may be **coalesced** into a single event with the latest known state. Most of the time, this is the right thing to do. For example, if you're dragging a big object across the screen, and the application can't repaint the object fast enough to keep up with your mouse, you don't want the mouse movements to accumulate in the queue, because then the object will lag behind the mouse pointer, diligently (and foolishly) following the same path your mouse did.

Sometimes, however, coalescing hurts. If you're sketching a freehand stroke with the mouse, and some of the mouse movements are coalesced, then the stroke may have straight segments at places where there should be a smooth curve. If application delays are bursty, then coalescing may hurt even if your application can usually keep up with the mouse.

7.10 Event Loop

- While application is running
  - Block until an event is ready
  - Get event from queue
  - (sometimes) Translate raw event into higher-level events
    - Generates double-clicks, characters, focus, enter/exit, etc.
    - Translated events are put into the queue
  - Dispatch event to target component
- Who provides the event loop?
  - High-level GUI toolkits do it internally (Java, VB, C#)
  - Low-level toolkits require application to do it (MS Win, Palm, SWT)

The event loop reads events from the queue and dispatches them to the appropriate components in the view hierarchy. On some systems (notably Microsoft Windows), the event loop also includes a call to a function that translates raw events into higher-level ones. On most systems, however, translation happens when the raw event is added to the queue, not when it is removed.

Every GUI program has an event loop in it somewhere. Some toolkits require the application programmer to write this loop (e.g., Win32); other toolkits have it built-in (e.g., Java Swing). Unfortunately, Java's event loop is written as essentially an infinite loop, so the event loop thread never cleanly exits. As a result, the normal clean way to end a Java program – waiting until all the threads are finished – doesn't work for GUI programs. The only way to end a Java GUI program is System.exit(). This despite the fact that Java best practices say not to use System.exit(), because it doesn't guarantee to garbage collect and run finalizers.

Swing lets you configure your application's main JFrame with EXIT_ON_CLOSE behavior, but this is just a shortcut for calling System.exit().

7.11 Event Dispatch and Propagation

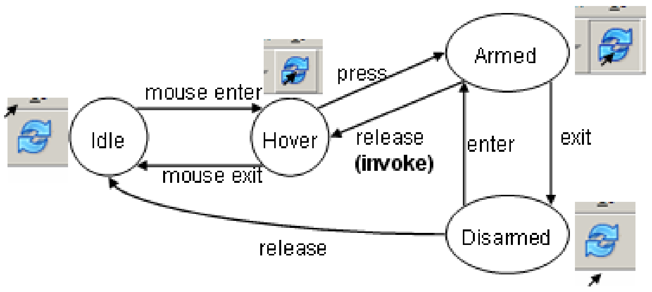
- Dispatch: choose target component for event
  - Key event: component with keyboard focus
  - Mouse event: component under mouse
    - **Mouse capture:** any component can grab mouse temporarily so that it receives all mouse events (e.g. for drag & drop)
- Propagation: if target component declines to handle event, the event passes up to its parent

Event dispatch chooses a component to receive the event. Key events are sent to the component with the keyboard focus, and mouse events are generally sent to the component under the mouse. An exception is **mouse capture**, which allows any component to grab all mouse events (essentially a mouse analogue for keyboard focus). Mouse capture is done automatically by Java when you hold down the mouse button to drag the mouse. Other UI toolkits give the programmer direct access to mouse capture – in the Windows API, for example, you'll find a SetMouseCapture function.

If the target component declines to handle the event, the event propagates up the view hierarchy until some component handles it. If an event bubbles up to the top without being handled, it is ignored.

7.12 Designing a Controller

- A controller is a finite state machine
- Example: push button



Now let's look at how components that handle input are typically structured. A controller in a direct manipulation interface is a **finite state machine**. Here's an example of the state machine for a push button's controller. **Idle** is the normal state of the button when the user isn't directing any input at it. The button enters the **Hover** state when the mouse enters it. It might display some feedback to reinforce that it affords clickability. If the mouse button is then pressed, the button enters the **Armed** state, to indicate that it's being pushed down. The user can cancel the button press by moving the mouse away from it, which goes into the **Disarmed** state. Or the user can release the mouse button while still inside the component, which invokes the button's action and returns to the **Hover** state. Transitions between states occur when a certain input event arrives, or sometimes when a timer times out. Each state may need different feedback displayed by the view. Changes to the model or the view occur on transitions, not states: e.g., a push button is actually invoked by the release of the mouse button.

7.13 Interactors

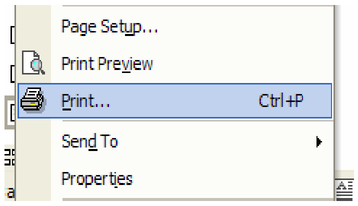
- Generic reusable controllers (Garnet and Amulet toolkits)
  - Selection interactor
  - Move/Grow interactor
  - New-point interactor
  - Text editing interactor
  - Rotating interactor
- Hide the details of handling input events and finite state machines
- Useful only in a component model
- Parameterized
  - start, stop, abort events
  - start location, inside/outside predicates
  - feedback components
  - callback procedures on event transitions

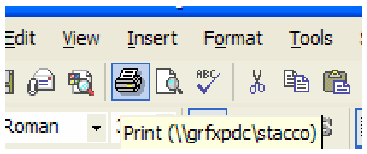
An alternative approach to handling low-level input events is the **interactor** model, introduced by the Garnet and Amulet research toolkits from CMU. Interactors are generic, reusable controllers, which encapsulate a finite state machine for a common task. They're mainly useful for the component model, in which the graphic output is represented by objects that the interactors can manipulate.

Lecture 8: Design Principles

8.1 UI Hall of Fame or Shame?

- Three ways to print in Microsoft Office
  - File/Print menu item
  - Print toolbar button
  - Ctrl-P keyboard shortcut





There are three ways to print in Microsoft Office applications: a menu command, a toolbar button, and a keyboard shortcut. That’s OK, because we want to provide shortcuts for experienced users (**flexibility & efficiency**).

Unfortunately, the three commands don’t all do the same thing:

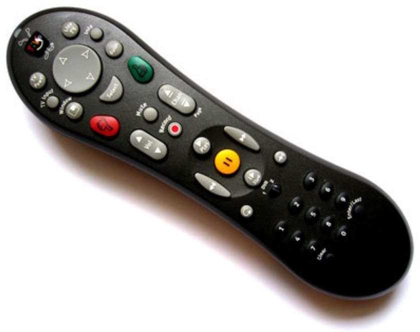
- File/Print brings up the Print dialog
- The Print toolbar button prints immediately using the latest print settings
- Ctrl-P brings up the Print dialog

So there are three commands named Print that do different things (**internal inconsistency**).

But the toolbar button’s behavior is useful! There *should* be an easy way to just print. Why? **Flexibility and efficiency**. Most of the time, for most users, for most copies of a document, you only want to print it one way. You don’t need to specify the printer (most people have only 1), you don’t need to pick color or grayscale or duplex or paper source. The default settings, or the last settings you chose for this particular document, should work. Don’t ask me all those questions, just give me a printout! That’s what the toolbar button is trying to do. But the fact that all three are simply named Print is disturbing. Furthermore, there’s a natural hierarchy among these shortcuts, starting with the menu item, which is clearly labeled knowledge in the world; then the toolbar button, which is always visible, faster to reach than the menu item, less descriptive than a label, but still allows you to recognize rather than recall; and finally the keyboard shortcut, which is mnemonic (Ctrl-P for Print) but requires knowledge in the head.

The print-now command is **unnaturally mapped** on this hierarchy. It’s mapped to the medium-shortcut toolbar button, but not to the extreme-shortcut Ctrl-P. As a result, I (for one) hesitate to use either of the print shortcuts, because I’m frequently surprised by what it does.

8.2 UI Hall of Fame or Shame?



Let's look at another example. Appliance remote controls are known for being complicated --- some of them bristle with tiny buttons, all alike. The Tivo remote shown here is noteworthy for its simplicity and careful attention to good UI design principles:

- simplicity! Doesn't have a million buttons on it. Most features are controlled by onscreen software. The remote only needs a pointing device and essential shortcuts.
- important buttons are large (Fitts's Law) and have unique shapes (consistency).
- related buttons are placed in natural mapping: e.g., the back/forward buttons. The channel and volume buttons are both mapped vertically – a natural mapping, but it makes the channel and volume buttons very similar to each other as a result. It's a tradeoff.
- the Pause button is large, indicating a good task analysis for the way Tivo is used. Pausing live TV is a big reason people buy Tivos. Why is the Play button small?
- great graphic design: a few simple colors used only for highlighting important controls; high contrast makes the labels easy to read
- good industrial design as well: the remote is shaped and balanced well for the user's hand.

One downside that many commentators have mentioned: it's too symmetrical. If you grasp it around the waist without looking, you can't tell which end to point at the TV, and you may end up fast-forwarding when you meant to rewind.

The New York Times had an interesting article (Feb 19, 2004) about the design that went into the remote. Lots of iteration, lots of prototypes:

http://www.nytimes.com/2004/02/19/technology/circuits/19remo.html?ex=1392526800&en=450d595187d25d27&ei=5007&partner=USERLAND

8.3 Usability Guidelines ("Heuristics")

- Plenty to choose from
  - Nielsen's 10 principles
    - One version in his book
    - A more recent version on his website
  - Tognazzini's 16 principles
  - Norman's rules from Design of Everyday Things
  - Mac, Windows, Gnome, KDE guidelines
- Help designers choose design alternatives
- Help evaluators find problems in interfaces ("heuristic evaluation")

Usability guidelines, or heuristics, are rules that distill out the principles of effective user interfaces. There are plenty of sets of guidelines to choose from – sometimes it seems like every usability researcher has their own set of heuristics. Most of these guidelines overlap in important ways, however. The experts don't disagree about what constitutes good UI. They just disagree about how to organize what we know into a small set of operational rules.

For the basis of this lecture, we'll use Jakob Nielsen's 10 heuristics, which can be found on his web site. (An older version of the same heuristics, with different names but similar content, can be found in his Usability Engineering book, one of the recommended books for this course.) Another good list is **Tog's First Principles** (find it in Google), 16 principles from Bruce Tognazzini that include affordances and Fitts's Law. In the last lecture, we talked about some design guidelines proposed by Norman: visibility, affordances, constraints, feedback, and so on.

Platform-specific guidelines are also important and useful to follow. Platform guidelines tend to be very specific, e.g. you should have a File menu, and there command called Exit on it (not Quit, not Leave, not Go Away). Following platform guidelines ensures consistency among different applications running on the same platform, which is valuable for novice and frequent users alike. However, platform guidelines are relatively limited in scope, offering solutions for only a few of the design decisions in a typical UI.

Heuristics can be used in two ways: during design, to choose among different alternatives; and during evaluation, to find and justify problems in interfaces.

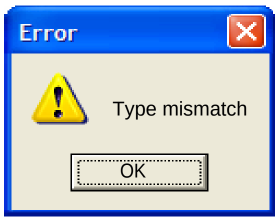
8.4 Guidelines From Earlier Lectures

- User-centered design
  - Know your users
  - Understand their tasks
- Fitts's Law
  - Size and proximity of controls should relate to their importance
  - Tiny controls are hard to hit
  - Screen edges are precious
- Memory
  - Use chunking to simplify information presentation
  - Minimize working memory
- Color guidelines
  - Don't depend solely on color distinctions (color blindness)
  - Avoid red on blue text (chromatic aberration)
  - Avoid small blue details
- Norman's principles of direct manipulation
  - Affordances
  - Natural mapping
  - Visibility
  - Feedback

Here are some guidelines we've already discussed in earlier lectures.

8.5 Match the Real World

- Use common words, not techie jargon
  - But use domain-specific terms where appropriate
- Don't put limits on user- defined names
- Allow aliases/synonyms in command languages
- Metaphors are useful but may mislead



Type mismatch

Let's look at each of Nielsen's 10 heuristics in detail.

First, the system should match the real world of the user's experience as much as possible. Nielsen's original name for this heuristic was "Speak the user's language", which is a good slogan to remember. 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.

If an interface allows users to name things, then users should be free to choose long, descriptive names. 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. Echoes of these limits persist in Windows even today.

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).

Metaphors are one way you can bring the real world into your 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.

8.6 Consistency and Standards

- Principle of Least Surprise
  - Similar things should look and act similar
  - Different things should look different
- Other properties
  - Size, location, color, wording, ordering, ...
- Command/argument order
  - Prefix vs. postfix
- Follow platform standards

The second heuristic is 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.

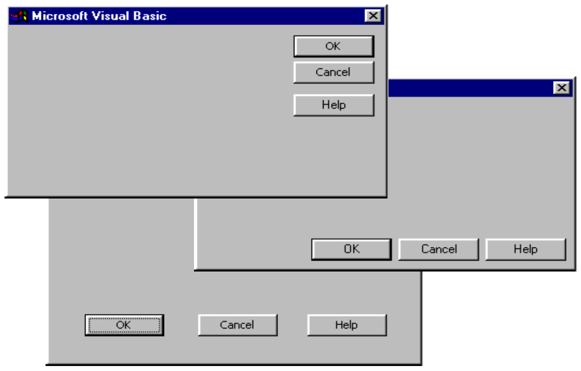
A very important kind of consistency is in wording. Use the same terms throughout your user interface. If your interface says “share price” in one place, “stock price” in another, and “stock quote” in a third, users will wonder whether these are three different things you’re talking about.

Incidentally, we’ve only looked at two heuristics, but already we have a contradiction! Matching the Real World argued for synonyms and aliases, so a command language should include not only delete but erase and remove too. But Consistency argues for only one name for each command, or else users will 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.

Command & argument ordering is another kind of consistency. In noun-verb order, the conventional order in graphical user interfaces, the user first selects the object of the command, and then invokes the command. In verb-noun order, the command is invoked first, and then the arguments are selected. A drawing program in which some commands were noun-verb and others were verb-noun would be very hard to learn and use.

8.7 Kinds of Consistency

- Internal
- External
- Metaphorical



There are three kinds of consistency you need to worry about: **internal consistency** within your application (like the VB dialog boxes shown); **external consistency** with other applications on the same platform (how do other Windows apps lay out OK and Cancel?); and **metaphorical consistency** with your interface metaphor or similar real-world objects.

We discussed the RealCD interface in an earlier lecture – it 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).

8.8 Case Against Consistency (Grudin)

- Inconsistency is appropriate when context and task demand it
  - Arrow keys
- But if all else is equal, consistency wins
  - QWERTY vs. Dvorak

Jonathan Grudin (in "The Case Against User Interface Consistency, CACM v32 n10, Oct 1989) finesses the issue of consistency still further. His argument is that consistency should not be treated as a sacred cow, but rather remain subservient to the needs of context and task. For example, although the inverted-T arrow-key arrangement on modern keyboards is both internally and metaphorically inconsistent in the placement of the down arrow, it's the right choice for efficiency of use. If two design alternatives are otherwise equivalent, however, consistency should carry the day.

Designs that are seriously inconsistent but provide only a tiny improvement in performance will probably fail. The Dvorak keyboard, for example, is slightly faster than the standard QWERTY keyboard, but not enough to overcome the power of an entrenched standard.

8.9 Help and Documentation

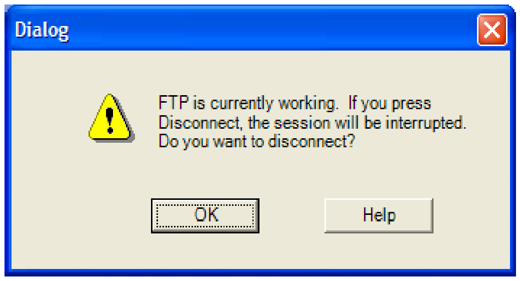
- Users don't read manuals
  - Prefer to spend time working toward their task goals, not learning about your system
- But manuals and online help are vital
  - Usually when user is frustrated or in crisis
- Help should be:
  - Searchable
  - Context-sensitive
  - Task-oriented
  - Concrete
  - Short

The next heuristic is (good) Help and Documentation. The sad fact about documentation is that most users simply don't read it, at least not before they try the interface. As a result, when they finally do want to look at the manual, it's because they've gotten stuck. Good help should take this into account.

A good point was raised in class that exclusively task-oriented help (which has largely taken over in Microsoft Windows) makes it impossible to get a high-level overview of an interface from the manual. So it's possible to go too far.

8.10 User Control and Freedom

- Provide undo
- Long operations should be cancelable
- All dialogs should have a cancel button



Dialog

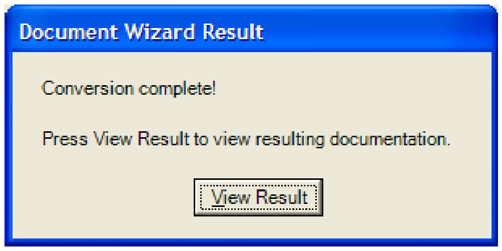
FTP is currently working. If you press Disconnect, the session will be interrupted. Do you want to disconnect?

This heuristic used to be called "Clearly Marked Exits" in Nielsen's old list. Users should not be trapped by the interface. Every dialog box should have a cancel button (where is it in this dialog box?), and long operations should be interruptible.

Users should be able to explore the interface without fear of being trapped in a corner. Undo is a great way to support exploration.

8.11 Visibility of System Status

- Keep user informed of system state
  - Cursor change
  - Selection highlight
  - Status bar
  - Don't overdo it...
- Response time
  - < 0.1 s: seems instantaneous
  - 0.1-1 s: user notices, but no feedback needed
  - 1-5 s: display busy cursor
  - > 1-5 s: display progress bar



Document Wizard Result

Conversion complete!

Press View Result to view resulting documentation.

This heuristic used to be called, simply, "Feedback." Keep the user informed about what's going on.

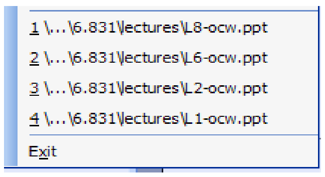
We've developed lots of idioms for feedback in graphical user interfaces. Use them:

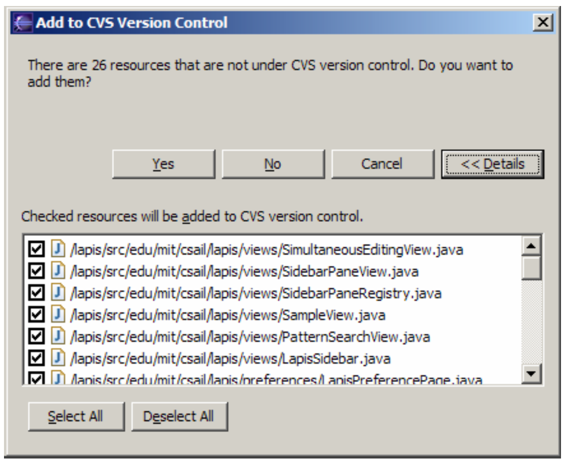
- Change the cursor to indicate possible actions (e.g. hand over a hyperlink), modes (e.g. drag/drop), and activity (hourglass).
- Use highlights to show selected objects. Don't leave selections implicit.
- Use the status bar for messages and progress indicators.

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. Depending on how long an operation takes, you may need different amounts of feedback. Even though we say "no feedback needed" if the operation takes less than a second, remember that something should change, visibly, within 100 ms, or perceptual fusion will be disrupted.

8.12 Flexibility and Efficiency

- Provide easily-learned shortcuts for frequent operations
  - Keyboard accelerators
  - Command abbreviations
  - Styles
  - Bookmarks
  - History


1 \...\6.831\lectures\L8-ocw.ppt
2 \...\6.831\lectures\L6-ocw.ppt
3 \...\6.831\lectures\L2-ocw.ppt
4 \...\6.831\lectures\L1-ocw.ppt
Exit


Add to CVS Version Control

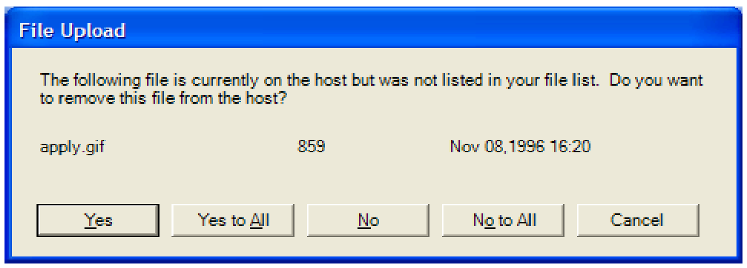
There are 26 resources that are not under CVS version control. Do you want to add them?

Yes No Cancel << Details

Checked resources will be added to CVS version control.

☑ /lapis/src/edu/mit/csail/lapis/views/SimultaneousEditingView.java
☑ /lapis/src/edu/mit/csail/lapis/views/SidebarPaneView.java
☑ /lapis/src/edu/mit/csail/lapis/views/SidebarPaneRegistry.java
☑ /lapis/src/edu/mit/csail/lapis/views/SampleView.java
☑ /lapis/src/edu/mit/csail/lapis/views/PatternSearchView.java
☑ /lapis/src/edu/mit/csail/lapis/views/LapisSidebar.java
☑ /lapis/src/edu/mit/csail/lapis/preferences/LapisPreferencePage.java

Select All Deselect All


File Upload

The following file is currently on the host but was not listed in your file list. Do you want to remove this file from the host?

apply.gif 859 Nov 08,1996 16:20

Yes Yes to All No No to All Cancel

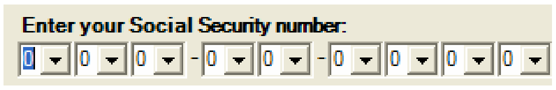
This heuristic used to be called "Shortcuts." Frequent users need and want them.

Recently-used history is one very useful kind of shortcut, like this recently-used files menu.

We looked at some other shortcuts in an earlier hall of fame & shame. Yes to All and No to All were good, but they don't smoothly handle the case where the user wants to choose a mix of Yes and No. Eclipse's list of checkboxes, with Select All and Deselect All, provides the right mix of flexibility and efficiency.

8.13 Error Prevention

- Selection is less error-prone than typing
  - But don't go overboard...



Enter your Social Security number:

0 0 0 - 0 0 - 0 0 0 0

- Disable illegal commands

Now we get into heuristics about error handling. Since humans make errors if they're given a chance (this is called Murphy's Law: "if something can go wrong, it will"), the best solution is to prevent errors entirely. One way to prevent errors is to allow users to select rather type. Misspellings then become impossible. This attitude can be taken to an extreme, however, as shown in this example.

If a command is illegal in the current state of the interface – e.g., Copy is impossible if nothing is selected - then the command should be disabled ("grayed out") so that it simply can't be selected in the first place.

8.14 Description Error

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

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

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 switches.

8.15 Capture Error

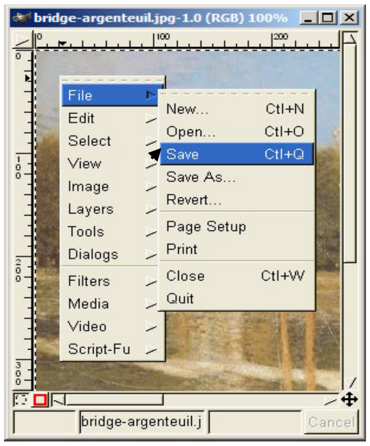
- A sequence of actions is replaced by another sequence that starts the same way
  - Leave your house and find yourself walking to school instead of where you meant to go
  - Vi :wq command
- Avoid habitual action sequences with common prefixes

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

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

8.16 Mode Error

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



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

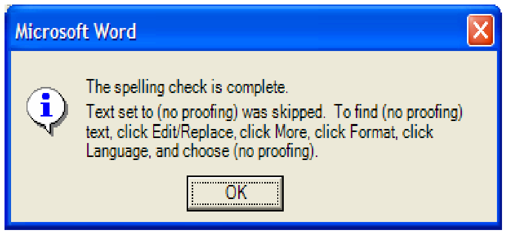
Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs. There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. When modes are necessary, it's essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn't actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user's locus of attention. That's why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn't really work. (Raskin, The Humane Interface, 2000 has a good discussion of locus of attention as it relates to mode visibility.)

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

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

8.17 Recognition, Not Recall

- Use menus, not command languages
- Use combo boxes, not textboxes
- Use generic commands where possible (Open, Save, Copy Paste)
- All needed information should be visible



The spelling check is complete.

Text set to (no proofing) was skipped. To find (no proofing) text, click Edit/Replace, click More, click Format, click Language, and choose (no proofing).

There's another reason why selection is better than typing – it reduces the user's memory load. "Minimize Memory Load" was the original name for this heuristic, and it drives much of modern user interface design.

Norman (in The Design of Everyday Things) makes a useful distinction between knowledge in the head, which is hard to get in there and still harder to recover, and knowledge in the world, which is far more accessible. Knowledge in the head is what we usually think of as knowledge and memory. Knowledge in the world, on the other hand, means not just documentation and button labels and signs, but also nonverbal features of a system that constrain our actions or remind us of what to do. Affordances, constraints, and feedback are all aspects of knowledge in the world. Command languages demand lots of knowledge in the head, while menus rely on knowledge in the world.

Generic commands are polymorphic, working the same way across a wide variety of data objects and applications. Generic commands are powerful because only one command has to be learned and remembered.

Any information needed by a task should be visible or otherwise accessible in the interface for that task. The interface shouldn't depend on users to remember the email address they want to send mail to, or the product code for the product they want to buy.

This dialog box is a great example of overreliance on the user's memory. It's a modal dialog box, so the user can't start following its instructions until after clicking OK. But then the instructions vanish from the screen, and the user is left to struggle to remember them. An obvious solution to this problem would be a button that simply executes the instructions directly! This message is clearly a last-minute patch for a usability problem.

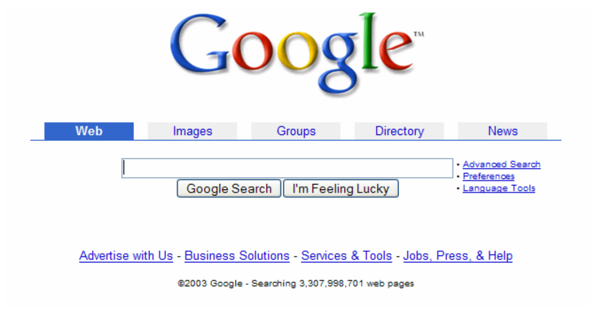
8.18 Error Reporting, Diagnosis, Recovery

- Be precise; restate user's input
  - Not "Cannot open file", but "Cannot open file named paper.doc"
- Give constructive help
  - why error occurred and how to fix it
- Be polite and nonblaming
  - Not "fatal error", not "illegal"
- Hide technical details (stack trace) until requested

If you can't prevent the error, give a good error message. A good error message should (1) be precise; (2) speak the user's language, avoiding technical terms and details unless explicitly requested; (3) give constructive help; and (4) 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.

8.19 Aesthetic and Minimalist Design

- "Less is More"
  - Omit extraneous info, graphics, features

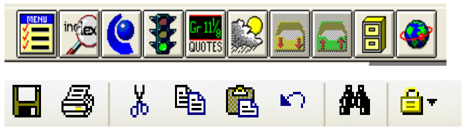

Google™
Web
Images
Groups
Directory
News
Google Search
I'm Feeling Lucky
• Advanced Search
• Preferences
• Language Tools
Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help
©2003 Google - Searching 3,307,998,701 web pages

The final heuristic is a catch-all for a number of rules of good graphic design, which really boil down to one word: simplicity. Leave things out unless you have good reason to include them. Don't put more help text on your main window than what's really necessary. Leave out extraneous graphics. Most important, leave out unnecessary features. If a feature is never used, there's no reason for it to complicate your interface.

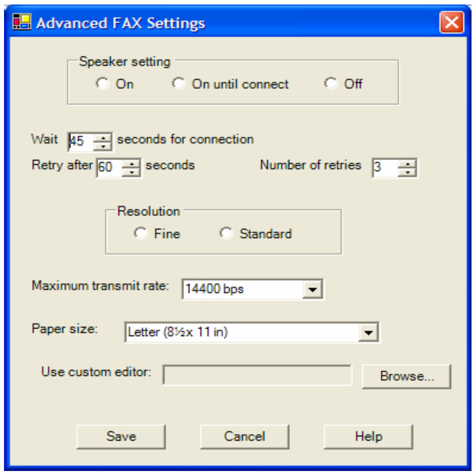
Google and the Tivo remote offer great positive examples of the less-is-more philosophy.

8.20 Aesthetic and Minimalist Design

- Good graphic design
  - Few, well-chosen colors and fonts



  - Group with whitespace
  - Align controls sensibly
- Use concise language
  - Choose labels carefully



Advanced FAX Settings

Speaker setting
On   On until connect   Off

Wait 45 seconds for connection
Retry after 60 seconds   Number of retries 3

Resolution
Fine   Standard

Maximum transmit rate: 14400 bps

Paper size: Letter (8½x 11 in)

Use custom editor: Browse...

Save   Cancel   Help

Use few, well-chosen colors. The toolbars at the top show the difference between cluttered and minimalist color design. The first toolbar is full of many saturated colors. It's not only gaudy and distracting, but actually hard to scan. The second toolbar uses only a handful of colors – black, white, gray, blue, yellow. It's muted, calming, and the few colors are used to great effect to distinguish the icons. The whitespace separating icon groups helps a lot too.

The dialog box shows how cluttered and incomprehensible a layout can look when controls aren't aligned. We'll look at graphic design in more detail in a future lecture.

8.21 Chunking the Heuristics Further

- Meet expectations
  1. Match the real world
  2. Consistency & standards
  3. Help & documentation
- User is the boss
  4. User control & freedom
  5. Visibility of system status
  6. Flexibility & efficiency
- Handle errors
  7. Error prevention
  8. Recognition, not recall
  9. Error reporting, diagnosis, and recovery
- Keep it simple
  10. Aesthetic & minimalist design

Since it's hard to learn 10 heuristics and hold them in your head when you're trying to design, I find it useful to categorize Nielsen's heuristics still further.

Meet expectations. The first three heuristics concern how well the interface fits its environment, its task, and its users: speaking the user's language, keeping consistent with itself and other applications, and satisfying the expectation of help when it's needed.

User is the boss. The next three heuristics are related in that the interface should serve the user, rather than the other way around. Don't push the boss into the corner, keep the boss aware of things, and make the boss productive and efficient.

Handle errors. The next three heuristics largely concern errors, which are part and parcel of human-computer interaction: prevent them as much as possible, don't rely on human memory, but when errors are unavoidable, report them properly.

Aesthetic & minimal design stays in its own category, as befits its overwhelming importance. **Keep it simple.**

8.22 Tog's 16 Principles

- **Anticipation**
- Autonomy
- Color blindness
- Consistency
- **Defaults**
- Efficiency
- **Explorable interfaces**
- Fitts's Law
- Human interface objects
- Latency reduction
- **Learnability**
- Metaphors
- **Protect users' work**
- **Readability**
- **Track state**
- **Visible navigation**

Let's look at a couple other lists of guidelines, because they highlight other good rules of design. Here is Bruce Tognazzini's list. We've seen most of these already; let's just focus on the few that are new, or that highlight particularly important problems to avoid.

Anticipation means that a good design should put all needed information and tools within the user's easy reach. Anticipation is the reason why a File Save dialog box needs a way to create a new folder. Note that you can't anticipate the user's needs without a thorough task analysis!

Defaults are common answers already filled into a form. Defaults help in lots of ways: they provide shortcuts to both novices and frequent users; they help the user learn the interface by showing examples of legal entries. But Tog advises that defaults should be *fragile*, coming up already selected so that frequent users can immediately overtype them. Tog also advises removing the actual word "default" from your interface's vocabulary, which makes sense because it has some very negative connotations in the lending world. **Explorable interfaces** is basically User Control and Freedom, but deserves special notice. One way users learn is by exploring: poking around an interface, trying things out. Your interface should encourage this kind of exploration, with physically reversible actions, undo, and cancel. 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.

Learnability is one of our usability criteria (along with efficiency, memorability, error rate, and satisfaction). *Every* user is a novice with your interface at some point. Design for learnability. Even if your target users are frequent users, who receive heavy direct training in using your interface, you can still make design decisions that make this learning easier.

Protect users' work is certainly error prevention, but it highlights an important value judgment: errors that lose or destroy the user's work are the worst kind. It's worth substantial engineering to prevent this from happening.

Readability is a graphic design question, but it's one of the most important aspects of graphic design in most user interfaces, whether desktop or web. Choose font size and color contrast to maximize the readability of text, particularly for aging users.

Track state is a kind of shortcut in the sense that you should save the user from restoring the state of their last session. Keep histories of users' previous choices; when you run the Print function again, remember the settings the user provided before.

Finally, **visible navigation** is a kind of visibility of system state. On the Web, in particular, users are in danger of getting lost. Help prevent this by visualizing the user's location; popular techniques include bread crumb trails (like Busiiness & Economy >> Finance & Investing >> Banking) and highlights in navigation bars.

8.23 Shneiderman's 8 Golden Rules

- Consistency
- Shortcuts
- Feedback
- **Dialog closure**
- Simple error handling
- Reversible actions
- Put user in control
- Reduce short-term memory load

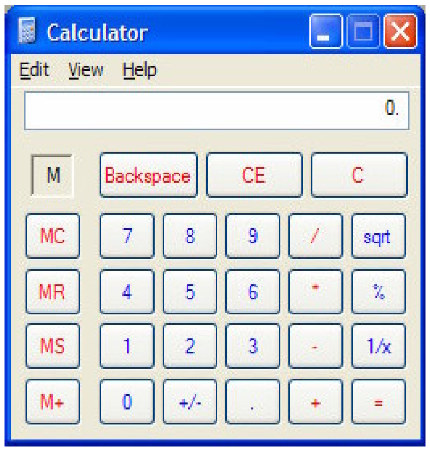
One more list: Shneiderman's 8 Golden Rules of UI design include most of the principles we've already discussed. The new one is **dialog closure**. Action sequences should be designed with a beginning, a middle, and an end. For example, think about drag and drop:

- At the beginning, you press the mouse button and see the object picked up with your cursor.
- In the middle, you move the object across the screen towards your target, getting feedback that it's coming along.
- At the end, you release the mouse button, and see the effects of the drop.

The key feature of closure is the feedback you get at the end of the operation. This assurance that the operation completed provides the user with a sense of accomplishment, some relief, and an opportunity to clear their working memory of the details of the task in preparation for another.

Lecture 9: Paper Prototyping

9.1 UI Hall of Fame or Shame?



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.

Unfortunately 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.

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 heuristic. 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.

9.2 Today’s Topics

- Paper prototypes
- Wizard of Oz prototypes

Today we’re going to talk about protototyping: producing cheaper, less accurate renditions of your target interface. Prototyping is essential in the early iterations of a spiral design process, and it’s useful in later iterations too.

9.3 W hy Prototype?

- Get feedback earlier, cheaper
- Experiment with alternatives
- Easier to change or throw away

We build prototypes for several reasons, all of which largely boil down to cost.

First, prototypes are much faster to build than finished implementations, so we can evaluate them sooner and get early feedback about the good and bad points of a design.

Second, if we have a design decision that is hard to resolve, we can build multiple prototypes embodying the different alternatives of the decision.

Third, if we discover problems in the design, a prototype can be changed more easily, for the same reasons it could be built faster. Prototypes are more malleable. Most important, if the design flaws are serious, a prototype can be thrown away. It’s important not to commit strongly to design ideas in the early stages of design. Unfortunately, writing and debugging a lot of code creates a psychological sense of commitment which is hard to break. You don’t want to throw away something you’ve worked hard on, so you’re tempted to keep some of the code around, even if it really should be scrapped.

The prototyping techniques we’ll see in this lecture actually force you to throw the prototype away. For example, a paper mockup won’t form any part of a finished software implementation. This is a good mindset to have in early iterations, since it maximizes your creative freedom.

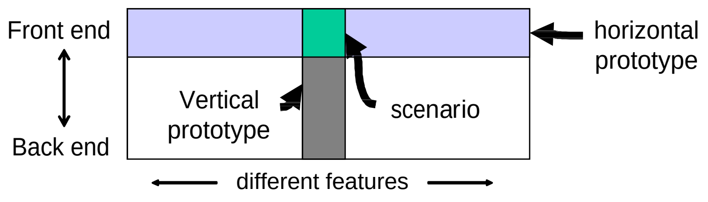
9.4 Prototype Fidelity

- Low fidelity: omits details
- High fidelity: more like finished product

An essential property of a prototyping technique is its fidelity, which is simply how similar it is to the finished interface. Low-fidelity prototypes omit details, use cheaper materials, or use different interaction techniques. High-fidelity prototypes are very similar to the finished product.

9.5 Fidelity is Multidimensional

- Breadth: % of features covered
  - Only enough features for certain tasks
- Depth: degree of functionality
  - Limited choices, canned responses, no error handling



Fidelity is not just one-dimensional, however. Prototypes can be low- or high-fidelity in various different ways (Carolyn Snyder, *Paper Prototyping*, 2003).

Breadth refers to the fraction of the feature set represented by the prototype. A prototype that is low-fidelity in breadth might be missing many features, having only enough to accomplish certain specific tasks. A word processor prototype might omit printing and spell-checking, for example.

Depth refers to how deeply each feature is actually implemented. Is there a backend behind the prototype that's actually implementing the feature? Low-fidelity in depth may mean limited choices (e.g., you can't print double-sided), canned responses (always prints the same text, not what you actually typed), or lack of robustness and error handling (crashes if the printer is offline).

A diagrammatic way to visualize breadth and depth is shown (following Nielsen, *Usability Engineering*, p. 94). A **horizontal prototype** is all breadth, and little depth; it's basically a frontend with no backend. A **vertical prototype** is the converse: one area of the interface is implemented deeply. The question of whether to build a horizontal or vertical prototype depends on what risks you're trying to mitigate. In user interface design, horizontal prototypes are more common, since they address usability risk. But if some aspect of the application is a risky implementation – you're not sure if it can be implemented to meet the requirements – then you may want to build a vertical prototype to test that.

A special case lies at the intersection of a horizontal and a vertical prototype. A **scenario** shows how the frontend would look for a single concrete task. Scenarios are great for visualizing a design, but they're hard to evaluate with users.

9.6 More Dimensions of Fidelity

- Look: appearance, graphic design
  - Sketchy, hand-drawn
- Feel: input method
  - Pointing & writing feels very different from mouse & keyboard

Two more crucial dimensions of a prototype's fidelity are, loosely, its look and its feel. Look is the appearance of the prototype. A hand-sketched prototype is low-fidelity in look, compared to a prototype that uses the same widget set as the finished implementation. Feel refers to the physical methods by which the user interacts with the prototype. A user interacts with a paper mockup by pointing at things to represent mouse clicks, and writing on the paper to represent keyboard input. This is a low-fidelity feel for a desktop application (but it may not be far off for a tablet PC application).

9.7 Paper Prototype

- Interac it ve paper mockup
  - Sketches of screen appearance
  - Paper pieces show windows, menus, dialog boxes
- Interaction is natural
  - Pointing with a finger = mouse click
  - Writing = typing
- A person simulates the computer's operation
  - Putti ng down & picking up pieces
  - Writing responses on the screen
  - Describing effects that are hard to show on paper
- Low fidelity in look & feel
- High fidelity in depth (person simulates the backend)

Paper prototypes are an excellent choice for early design iterations. A paper prototype is a physical mockup of the interface, mostly made of paper. It's usually hand-sketched on mutiple pieces, with different pieces showing different menus, dialog boxes, or window elements.

The key difference between mere sketches and a paper prototype is **interactivity.** A paper prototype is brought to life by a design team member who simulates what the computer would do in response to the user's "clicks" and "keystrokes", by rearranging pieces, writing custom responses, and occasionally announcing some effects verbally that are too hard to show on paper. Because a paper prototype is actually interactive, you can actually user-test it: give users a task to do and watch how they do it.

A paper prototype is clearly low fidelity in both look and feel. But it can be arbitrarily high fidelity in breadth at very little cost (just sketching, which is part of design anyway). Best of all, paper prototypes can be **high-fidelity in depth** at little cost, since a human being is simulating the backend.

9.8 W hy Paper Prototyping?

- Faster to build
  - Sketching is faster than programming
- Easier to change
  - Easy to make changes between user tests, or even during a user test
  - No code investment everything will be thrown away (except the design)
- Focuses attention on big picture
  - Designer doesn�similart waste time on details
  - Customer makes more creative suggestions, not nitpicking
- Nonprogrammers can help
  - Only kindergarten skills are required

But why use paper? And why hand sketching rather than a clean drawing from a drawing program?

Hand-sketching on paper is faster. You can draw many sketches in the same time it would take to draw one user interface with code. For most people, hand-sketching is also faster than using a drawing program to create the sketch.

Paper is easy to change. You can even change it during user testing. If part of the prototype was a problem for one user, you can scratch it out or replace it before the next user arrives. Surprisingly, paper is more malleable than digital bits in many ways.

Hand-sketched prototypes in particular are valuable because they focus attention on the issues that matter in early design without distracting anybody with details. When you’re sketching by hand, you aren’t bothered with details like font, color, alignment, whitespace, etc. In a drawing program, you would be faced with all these decisions, and you might spend a lot of time on them – time that would clearly be wasted if you have to throw away this design. Hand sketching also improves the feedback you get from users. They’re less likely to nitpick about details that aren’t relevant at this stage. They won’t complain about the color scheme if there isn’t one. More important, however, a hand-sketch design seems less finished, less set in stone, and more open to suggestions and improvements. Architects have known about this phenomenon for many years. If they show clean CAD drawings to their clients in the early design discussions, the clients are less able to discuss needs and requirements that may require radical changes in the design. In fact, many CAD tools have an option for rendering drawings with a “sketchy” look for precisely this reason.

A final advantage of paper prototyping: no special skills are required. So graphic designers, usability specialists, and even users can help create prototypes and operate them.

8.9 Tools for Paper Prototyping

- White poster board (11″ x14″)
  - For background, window frame
- Big (unlined) index cards (4″ x 6″ , 5″ x8″)
  - For menus, window contents, and dialog boxes
- Restickable glue
  - For keeping pieces fixed
- White correction tape
  - For text fields, checkboxes, short messages
- Overhead transparencies
  - For highlighting, user ‶typing″
- Photocopier
  - For making multiple blanks

- Pens & markers, scissors, tape.

Here are the elements of a paper prototyping toolkit.

Although standard (unlined) paper works fine, you'll get better results from sturdier products like poster board and index cards. Use poster board to draw a static background, usually a window frame. Then use index cards for the pieces you'll place on top of this background. You can cut the index cards down to size for menus and window internals. Restickable Post-it Note glue, which comes in a roll-on stick, is a must. This glue lets you make all of your pieces sticky, so they stay where you put them. You can find this glue at Pearl Arts in Central Square; it's not found in the Coop.

Post-it correction tape is another essential element. It's a roll of white tape with Post-it glue on one side. Correction tape is used for text fields, so that users can write on the prototype without changing it permanently. You peel off a length of tape, stick it on your prototype, let the user write into it, and then peel it off and throw it away. Correction tape comes in two widths, "2 line" and "6 line". The 2-line width is good for single-line text fields, and the 6-line width for text areas. You can get correction tape at the Office Max in East Cambridge.

Overhead transparencies are useful for two purposes. First, you can make a selection highlight by cutting a piece of transparency to size and coloring it with a transparency marker. Second, when you have a form with several text fields in it, it's easier to just lay a transparency over the form and let the users write on that, rather than sticking a piece of correction tape in every field.

If you have many similar elements in your prototype, a photocopier can save you time. And, of course, the usual kindergarten equipment: pens, markers, scissors, tape.

8.10 Tips for G ood Paper Prototypes

- Make it larger than life
- Make it monochrome
- Replace tricky visual feedback with audible descriptions
  - Tooltips, drag & drop, animation, progress bar
- Keep pieces organized
  - Use folders & open envelopes

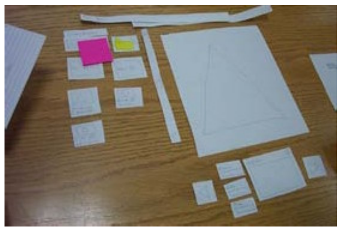
A paper prototype should be larger than life-size. Remember that fingers are bigger than a mouse pointer, and people usually write bigger than 12 point. So it'll be easier to use your paper prototype if you scale it up a bit. It will also be easier to see from a distance, which is important because the prototype lies on the table, and because when you're testing users, there may be several observers taking notes who need to see what's going on. Big is good. Don't worry too much about color in your prototype. Use a single color. It's simpler, and it won't distract attention from the important issues. Needless to say, don't use yellow.

You don't have to render every visual effect in paper. Some things are just easier to say aloud: "the basketball is spinning." "A progress bar pops up: 20%, 50%, 75%, done." If your design supports tooltips, you can tell your users just to point at something and ask

"What's this?", and you'll tell them what the tooltip would say. If you actually want to test the tooltip messages, however, you should prototype them on paper.

Figure out a good scheme for organizing the little pieces of your prototype. One approach is a three-ring binder, with different screens on different pages. Most interfaces are not sequential, however, so a linear organization may be too simple. Two-pocket folders are good for storing big pieces, and letter envelopes (with the flap open) are quite handy for keeping menus.

8.11 Hand-Drawn or Not?

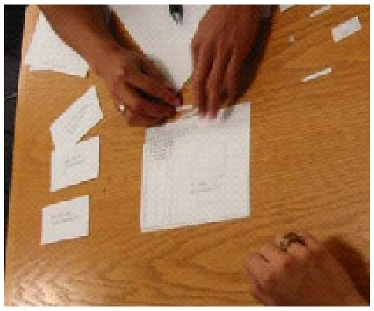


Here are some of the prototypes made by an earlier class. Should a paper prototype be hand-sketched or computer-drawn? Generally hand-sketching is better in early design, but sometimes realistic images can be constructive additions. Top left is a prototype for an interface that will be integrated into an existing program (IBM Eclipse), so the prototype is mostly constructed of modified Eclipse screenshots. The result is very clean and crisp, but

also tiny – it's hard to read from a distance. It may also be harder for a test user to focus on commenting about the new parts of the interface, since the new features look just like Eclipse. A hybrid hand-sketched/screenshot interface might work even better.

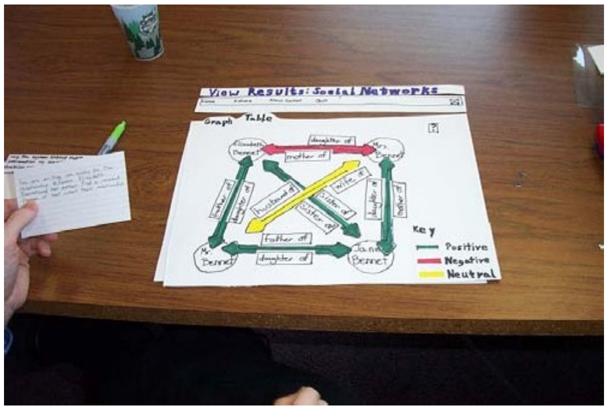
The top right prototype shows such a hybrid – a interface designed to integrate into a web browser. Actual screenshots of web pages are used, mainly as props, to make the prototype more concrete and help the user visualize the interface better. Since web page layout isn't the problem the interface is trying to solve, there's no reason to hand-sketch a web page. The bottom photo shows a pure hand-sketched interface that might have benefited from such props -- a photo organizer could use real photographs to help the user think about what kinds of things they need to do with photographs. This prototype could also use a window frame – a big posterboard to serve as a static background.

8.12 Size Matters



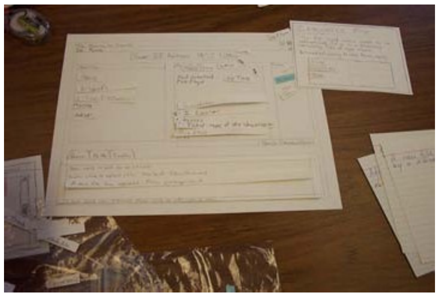
Both of these prototypes have good window frames, but the big one on the right is easier to read and manipulate.

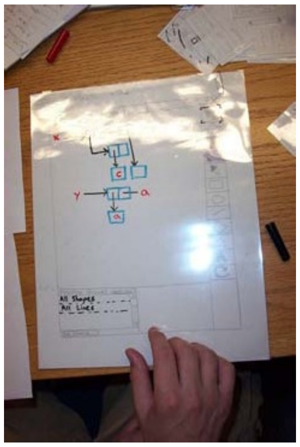
8.13 The Importance of Writing Big and Dark



This prototype is even easier to read. Markers are better than pencil. (Whiteout and correction tape can fix mistakes as well as erasers can!) Color is also neat, but don't bother unless color is a design decision that needs to be tested, as it is in this prototype. If color doesn't really matter, monochromatic prototypes work just as well.

8.14 Post-it Glue and Transparencies are Good





The prototype on the left has lots of little pieces that have trouble staying put. Post-it glue can help with that.

On the right is a prototype that's completely covered with a transparency. Users can write on it directly with dry-erase marker, which just wipes off – a much better approach than water-soluble transparency markers. With multiple layers of transparency, you can let the user write on the top layer, while you use a lower layer for computer messages, selection highlighting, and other effects.

8.15 How to Test a Paper Prototype

- Roles for design team
  - Computer
    - Simulates prototype
    - Doesn't give any feedback that the computer wouldn't
  - Facilitator
    - Presents interface and tasks to the user

    - Encourages user to "think a loud" by asking questions
    - Keeps user test from getting off track
  - Observer
    - Keeps mouth shut, sits on hands if necessary
    - Takes copious notes

Once you've built your prototype, you can put it in front of users and watch how they use it. We'll see much more about user testing in a later lecture, including ethical issues. But here's a quick discussion of user testing in the paper prototyping domain.

There are three roles for your design team to fill:

The **computer** is the person responsible for making the prototype come alive. This person moves around the pieces, writes down responses, and generally does everything that a real computer would do. In particular, the computer should not do anything that a real computer wouldn't. Think mechanically, and respond mechanically.

The **facilitator** is the human voice of the design team and the director of the testing session. The facilitator explains the purpose and process of the user study, obtains the user's informed consent, and presents the user study tasks one by one. While the user is working on a task, the facilitator tries to elicit verbal feedback from the user, particularly encouraging the user to "think aloud" by asking probing (but not leading) questions. The facilitator is responsible for keeping everybody disciplined and the user test on the right track.

Everybody else in the room (aside from the user) is an **observer**. The most important rule about being an observer is to keep your mouth shut and watch. Don't offer help to the user, even if they're missing something obvious. Bite your tongue, sit on your hands, and just watch. The observers are the primary note takers, since the computer and the facilitator are usually too busy with their duties.

8.16 What You Can Learn from a Paper Prototype

- Conceptual model
  - Do users understand it?
- Functionality
  - Does it do what's needed? Missing features?
- Navigation & task flow
  - Can users find their way around?
  - Are information precon id tions met?
- Terminology
  - Do users understand labels?
- Screen contents
  - What needs to go on the screen?

Paper prototypes can reveal many usability problems that are important to find in early stages of design. Fixing some of these problems require large changes in design. If users don't understand the metaphor or conceptual model of the interface, for example, the entire interface may need to be scrapped.

8.17 What You Can't Learn

- Look: color, font, whitespace, etc
- Feel: Fitts's Law issues
- Response time
- Are small changes noticed?
  - Even the tiniest change to a paper prototype is clearly visible to user
- Exploration vs. deliberation
  - Users are more d le iberate with a paper prototype; they don't explore or thrash as much.

But paper prototypes don't reveal every usability problem, because they are low-fidelity in several dimensions. Obviously, graphic design issues that depend on a high-fidelity look will not be discovered. Similarly, interaction issues that depend on a high-fidelity feel will also be missed. For example, Fitts's Law problems like buttons that are too small, too close together, or too far away will not be detected in a paper prototype.

The human computer of a paper prototype rarely reflects the speed of an implemented backend, so issues of response time – whether feedback appears quickly enough, or whether an entire task can be completed within a certain time constraint -- can't be tested either.

Paper prototypes don't help answer questions about whether subtle feedback will even be noticed. Will users notice that message down in the status bar, or the cursor change, or the highlight change? In the paper prototype, even the tiniest change is grossly visible, because a person's arm has to reach over the prototype and make the change. (If many changes happen at once, of course, then some of them may be overlooked even in a paper prototype, a clearly discernible. This is related to an interesting cognitive phenomenon called change blindness.)

There's an interesting qualitative distinction between the way users use paper prototypes and the way they use real interfaces. Experienced paper prototypers report that users are more deliberate with a paper prototype, apparently thinking more carefully about their actions. This may be partly due to the simulated computer's slow response; it may also be partly a social response, conscientiously trying to save the person doing the simulating from a lot of tedious and unnecessary paper shuffling. More deliberate users make fewer mistakes, which is bad, because you want to see the mistakes. Users are also less likely to randomly explore a paper prototype.

These drawbacks don't invalidate paper prototyping as a technique, but you should be aware of them. Several studies have shown that low-fidelity prototypes identify substantially the same usability problems as high- fidelity prototypes (Virzi, Sokolov, & Karis, "Usability problem identification using both low- and hi-fidelity prototypes", CHI '96; Catani & Biers, "Usability evaluation and prototype fidelity", Human Factors & Ergonomics 1998).

8.18 Wizard of Oz Prototype

- Software simulation with a human in the loop to help
- "Wizard of Oz" = "man behind the curtain"
  - Wizard is usually but not always hidden
- Often used to simulate future technology
  - Speech recognItion
  - Learning
- Issues
  - Two UIs to worry about: user's and wizard's
  - Wizard has to be mechanical

Part of the power of paper prototypes is the depth you can achieve by having a human simulate the backend. A Wizard of Oz prototype also uses a human in the backend, but the frontend is an actual computer system instead of a paper mockup. The term Wizard of Oz comes from the movie of the same name, in which the wizard was a man hiding behind a curtain, controlling a massive and impressive display.

In a Wizard of Oz prototype, the "wizard" is usually but not always hidden from the user. Wizard of Oz prototypes are often used to simulate future technology that isn't available yet, particularly artificial intelligence. A famous example was the listening typewriter (Gould, Conti, & Hovanyecz, "Composing letters with a simulated listening typewriter," CACM v26 n4, April 1983). This study sought to compare the effectiveness and acceptability of isolated-word speech recognition, which was the state of the art in the early 80's, with continuous speech recognition, which wasn't possible yet. The interface was a speech-operated text editor. Users looked at a screen and dictated into a microphone, which was connected to a typist (the wizard) in another room. Using a keyboard, the wizard operated the editor showing on the user's screen.

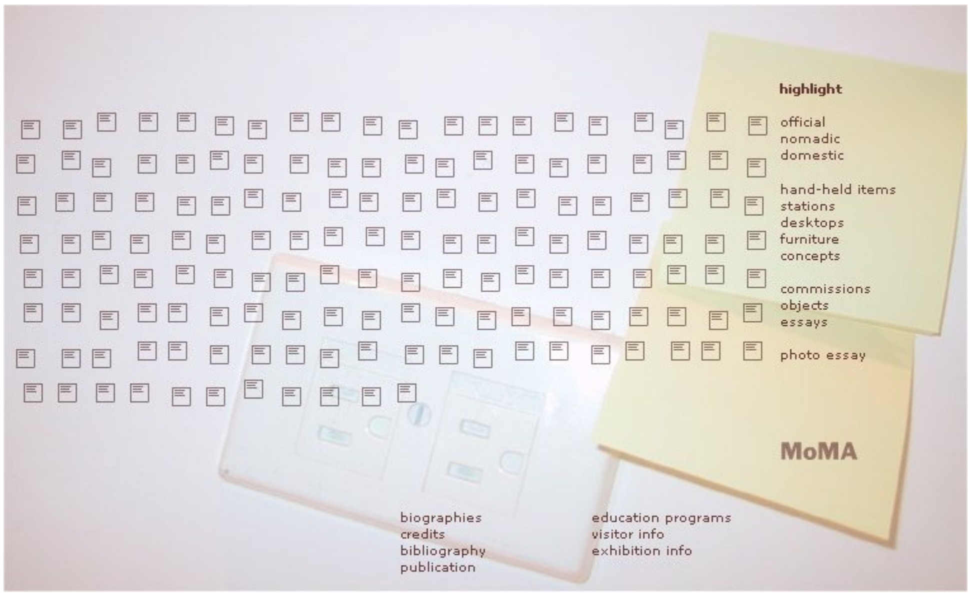
The wizard's skill was critical in this experiment. She could type 80 wpm, she practiced with the simulation for several weeks (with some iterative design on the simulator to improve her interface), and she was careful to type exactly what the user said, even exclamations and parenthetical comments or asides. The computer helped make her responses a more accurate simulation of computer speech recognition. It looked up every word she typed in a fixed dictionary, and any words that were not present were replaced with X’s, to simulate misrecognition. Furthermore, in order to simulate the computer’s ignorance of context, homophones were replaced with the most common spelling, so “done” replaced “dun”, and “in” replaced “inn”. The result was an extremely effective illusion. Most users were surprised when told (midway through the experiment) that a human was listening to them and doing the typing.

Thinking and acting mechanically is harder for a wizard than it is for a paper prototype simulator, because the tasks for which Wizard of Oz testing is used tend to be more “intelligent”. It helps if the wizard is personally familiar with the capabilities of similar interfaces, so that a realistic simulation can be provided. (See Maulsby, 1993) It also helps if the wizard’s interface can intentionally dumb down the responses, as was done in the Gould study.

A key challenge in designing a Wizard of Oz prototype is that you actually have two interfaces to worry about: the user’s interface, which is presumably the one you’re testing, and the wizard’s.

Lecture 10: Constraints and Layout

10.1 UI Hall of Fame or Shame?



http://www.moma.org/exhibitions/2001/workspheres/swfs/workspheres.html

This Flash-driven web site is the Museum of Modern Art's Workspheres exhibition, 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.

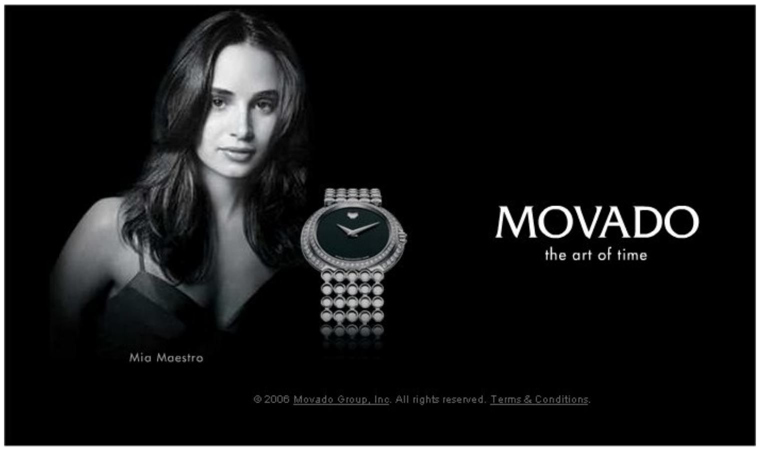
We can praise this site for at least one reason: incredible **simplicity**. The designer of this site was clearly striving for aesthetic appeal. Nothing unnecessary was included. Note the use of whitespace to group the list of categories on the right, and the simple heading *highlight* that gives a clue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

10.2 More "Mystery Navigation"


MOVADO
the art of time
Mia Maestro
© 2006 Movado Group, Inc. All rights reserved. Terms & Conditions.

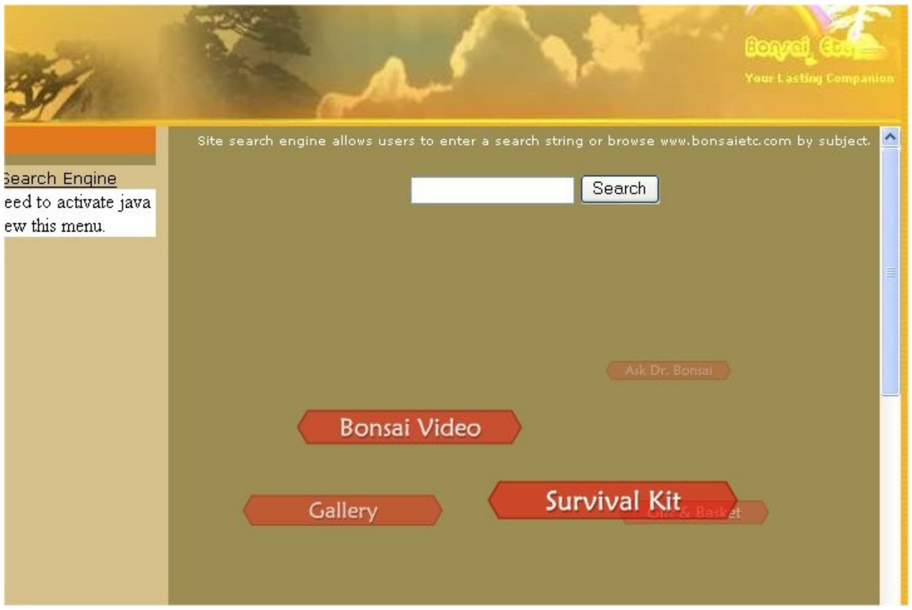
http://www.movado.com/

This is the home page for Movado, a company that makes expensive, stylish watches. The little white dots at the top of the window are menu options. If you watched the opening animation that precedes this screen, you'd see each menu label appear briefly over each dot. But if you skipped over the intro, you wouldn't see that, and you may not even realize that a menu is hiding up there under those stylish white dots.

When you mouse over a dot, you actually have to wait for a cute little animation (a watch hand sweeping around the dot) before the menu label appears. Each little animation takes 2 seconds. So scanning the entire menu to look at all the options takes 16 seconds!

Clearly this is even worse than MOMA's approach, since it starts with an invisible menu interface and makes it **inefficient** to boot. More tellingly, MOMA only cares about your eyeballs, but Movado actually wants to sell you a watch. If you can't figure out their menu, or lose patience with it, you may be headed elsewhere.

10.3 Let's Play a Menu Game



http://www.bonsaietc.com/BEtcSiteSearchEngine_Frame.htm

Here's our last entry: Bonsai Etc, a website that sells bonsai trees and equipment. In this site's Flash animation, the menu options **move**: some horizontally, some vertically. Worst of all, their paths overlap, so the items pass each other on the screen. At least they're labeled. It's a fun game, for a little while. But if you have a serious reason for visiting this web site – say, spending some money – do you really want to chase down every menu option you want to click?

One lesson you might draw from these examples is that Flash animation is bad, but that's too simplistic. Flash is a powerful tool that can be used for good or ill.

A better lesson might be that aesthetic appeal does not automatically confer usability. Effective graphic design is an important element of usability, but it isn't the whole story by any means.

10.4 Today's Topics

- Automatic layout
- Constraints

10.5 Layout

- Determining the positions and sizes of graphical objects

10.6 Layout Ranges in Difficulty

- Fixed constants
  - Many Windows dialog boxes
- Directly computable from model
  - Checkerboard from PS2/PS3
- One pass algorithm
  - Java layout managers, HTML tables
- Dynamic programming
  - paragraph flow with hyphenation
- Nonlinear optimization
- NP-hard
  - Graph with fewest edge crossings

10.7 Reasons to Do Layout Automatically

- Window resizing
- Screen resolution
- Font changes
- Widget changes
- Internationalization

10.8 Layout Managers

- Also called geometry managers (Tk, Motif)
- Abstract
  - Represents a bundle of constraint equations
- Local
  - Involve only the children of one container in the view hierarchy

10.9 Layout Propagation Algorithm

- layout(Container parent, Rectangle parentSize)
  - for each child in parent,
    - get child’s size request
  - apply layout constraints to fit children into parentSize
  - for each child,
    - set child’s size and position

10.10 Kinds of Layout Managers

- Packing
  - one dimensional
  - Tk: pack
  - Java: BorderLayout, FlowLayout, BoxLayout
- Gridding
  - two dimensional
  - Tk: grid
  - Java: GridLayout, GridBagLayout, TableLayout
- General
  - Java: SpringLayout

10.11 Important Concepts

- Anchoring
- Expanding vs. padding
- Invisible components
  - Struts
  - Glue
  - Springs
- Nested containers

10.12 Hints for Layout

- Use packing layouts when alignments are 1D
  - borders for top-level
  - nested boxes for internal
- Reserve gridding layouts for 2D alignment
  - unfortunately common when fields have captions!
  - TableLayout is easier than GridBag

10.13 Constraints

- Constraint: relationship expressed by the programmer and automatically maintained by the UI toolkit
- Uses
  - Layout
    - field.left = label.right + 10
  - Value propagation
    - deleteAction.enabled = (selection != null)
  - Synchronization of views to models
  - Interaction
    - rect.corner = mouse

10.14 One-Way Constraints

- Also called formulas, after spreadsheet
  - y = f(x1, x2, x3, ...)
  - Y depends on (points to) x1, x2, x3, ...
- Algorithms
  - Data-driven
    - Reevaluate formulas when a value is changed
  - Demand-driven
    - Reevaluate formulas whenever a value is requested
  - Lazy
    - When dependent value changes, **invalidate** all values that depend on it
    - When invalid value is requested, **recalculate** it

10.15 Variants

- Multi-output formulas
  - (y1, y2, ...) = f (x1, x2, x3, ...)
- Cyclic dependencies
  - Detect cycles and break them
- Constraint hierarchies
  - Some constraints stronger than others
- Side effects
  - If f has side effects, when do they happen?
    - Lazy evaluation makes side effects unpredictable
  - Amulet: eager evaluation

10.16 Multiway Constraints

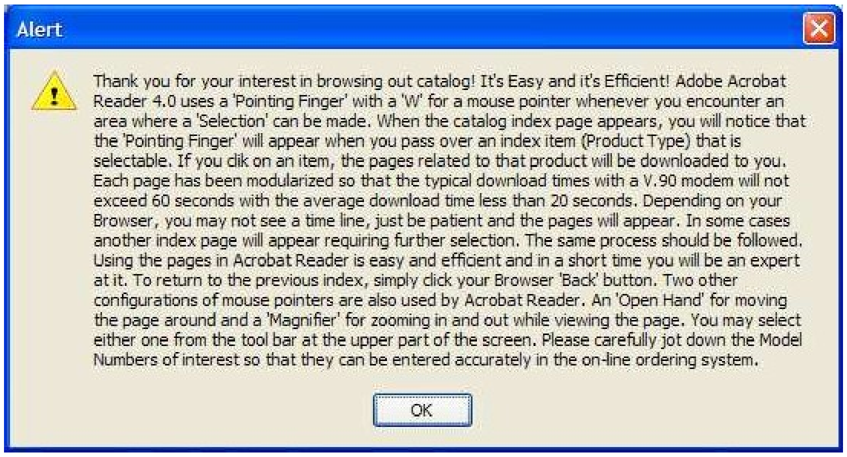
- Each constraint is a multivariate relationship
  - rect.right = rect.left + rect.width – 1
  - Any variable may be used as target (different method for each target variable)
  - Planning step decides which variables to target

10.17 Variants

- Constraint hierarchy
  - Which value should be changed?
  - Each constraint has a priority
  - "Stay" constraints (highest priority) are used for constants
- Inequalities
  - Label.right <= field.left

Lecture 11: Graphic Design

11.1 UI Hall of Fame or Shame?



Thank you for your interest in browsing out catalog! It's Easy and it's Efficient! Adobe Acrobat Reader 4.0 uses a 'Pointing Finger' with a 'W' for a mouse pointer whenever you encounter an area where a 'Selection' can be made. When the catalog index page appears, you will notice that the 'Pointing Finger' will appear when you pass over an index item (Product Type) that is selectable. If you clik on an item, the pages related to that product will be downloaded to you. Each page has been modularized so that the typical download times with a V.90 modem will not exceed 60 seconds with the average download time less than 20 seconds. Depending on your Browser, you may not see a time line, just be patient and the pages will appear. In some cases another index page will appear requiring further selection. The same process should be followed. Using the pages in Acrobat Reader is easy and efficient and in a short time you will be an expert at it. To return to the previous index, simply click your Browser 'Back' button. Two other configurations of mouse pointers are also used by Acrobat Reader. An 'Open Hand' for moving the page around and a 'Magnifier' for zooming in and out while viewing the page. You may select either one from the tool bar at the upper part of the screen. Please carefully jot down the Model Numbers of interest so that they can be entered accurately in the on-line ordering system.

Once upon a time, this bizarre help message was popped up by a website (Midwest Microwave) when users requested to view the site's product catalog. The message appears before the catalog is displayed. Clearly this message is a patch for usability problems in the catalog itself, but the patch itself has so many usability issues that it's worth listing a few.

- Overwhelming the user with detail. What's important here, and what isn't? (**minimalist desig**n)
- Horrible layout: no paragraphs, no headings, no whitespace to guide the eye (**aethestic design**)
- No attempt to organize the material into chunks so that it can be scanned, to find out what the user doesn't already know (**flexibility and efficiency**)
- This information is useless and out of context before the user has seen the task they'll be faced with (**help and documentation**)
- It's a modal dialog box, so all this information will go away as soon as the user needs to get to the catalog (**minimize memory load**)
- Using technical terms like V.90 modem (**speak the user's language**)
- "Please carefully jot down the Model Numbers" (**recognition, not recall**)
- Poor response times: 20-60 second response times (**user control and freedom**), though in fairness this was common for the web at the time, and maybe Acrobat has sufficient progress interfaces to make up for it.
- Misspelling "our catalog" in the first line (**speak the user's language**, but really we don't need a heuristic to justify pointing out a spelling errors)

11.2 Guideline for Good Graphic Design

- Simplicity
- Contrast
- White space
- Balance
- Alignment

Today, we're going to look at some specific guidelines for graphic design. These guidelines are drawn from the excellent book *Designing Visual Interfaces* by Kevin Mullet and Darrell Sano (Prentice-Hall, 1995). Mullet & Sano's book predates the Web, but the principles it describes are timeless and relevant to any visual medium.

Another excellent book is Edward Tufte's *The Visual Display of Quantitative Information*. Some of the examples in this lecture are inspired by Tufte.

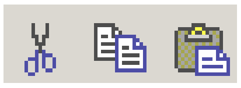
11.3 Simplicity

- "Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away." (Antoine de St- Exupery)
- "Simplicity does not mean the absence of any decor... It only means that the decor should belong intimately to the design proper, and that anything foreign to it should be taken away." (Paul Jacques Grillo)
- "Keep it simple, stupid." (KISS)
- "Less is more."
- "When in doubt, leave it out."

Okay, we'll shout some slogans at you now. You've probably heard some of these before. What you should take from these slogans is that designing for simplicity is a process of elimination, not accretion. Simplicity is in constant tension with task analysis, information preconditions, and other design guidelines, which might otherwise encourage you to pile more and more elements into a design, "just in case." Simplicity forces you to have a good reason for everything you add, and to take away anything that can't survive hard scrutiny.

11.4 Techniques for Simplicility: Reduction

- Remove inessential elements



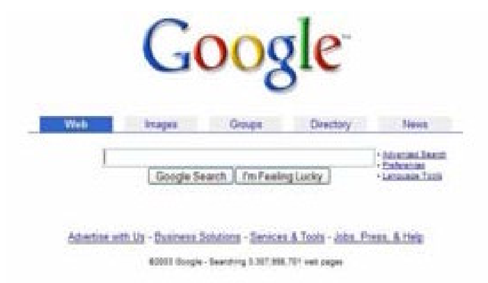


Image courtesy of Google. Used with permission.

Here are three ways to make a design simpler.

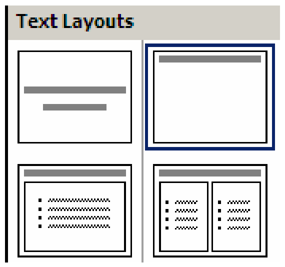
Reduction means that you eliminate whatever isn't necessary. This technique has three steps: (1) decide what essentially needs to be conveyed by the design; (2) critically examine every element (label, control, color, font, line weight) to decide whether it serves an essential purpose; (3) remove it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

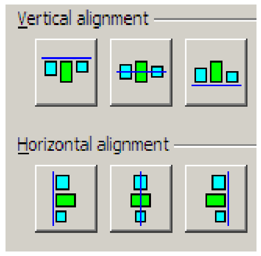
Icons demonstrate the principle of reduction well. A photograph of a pair of scissors can't possibly work as a 32x32 pixel icon; instead, it has to be a carefully-drawn picture which includes the bare minimum of details that are essential to scissors: two lines for the blades, two loops for the handles. The standard US Department of Transportation symbol for handicapped access is likewise a marvel of reduction. No element remains that can be removed from it without destroying its meaning.

We've already discussed the minimalism of Google and the Tivo remote in earlier classes. Here, the question is about functionality. Both Google and Tivo aggressively removed functions from their primary interfaces.

11.5 Techniques for Simplicility: Regularity

- Use a regular pattern
- Limit inessential variation among elements



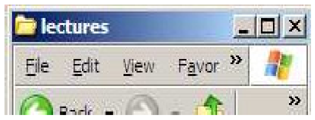


For the essential elements that remain, consider how you can minimize the unnecessary differences between them with **regularity**. Use the same font, color, line width, dimensions, orientation for multiple elements. Irregularities in your design will be magnified in the user's eyes and assigned meaning and significance. Conversely, if your design is mostly regular, the elements that you do want to highlight will stand out better. PowerPoint's Text Layouts menu shows both reduction (minimalist icons representing each layout) and regularity. Titles and bullet lists are shown the same way.

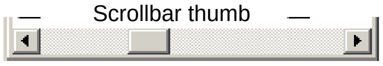
11.6 Techniques for Simplicility: Double-Duty

- Combine elements for leverage
  - Find a way for one element to play multiple roles

Title bar



Scrollbar thumb

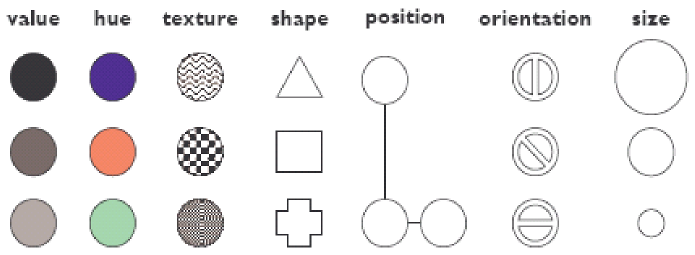


Help prompt

Finally, you can **combine elements**, making them serve multiple roles in the design. The desktop interface has a number of good examples of this kind of design. For example, the "thumb" in a scroll bar actually serves three roles. It affords dragging, indicates the position of the scroll window relative to the entire document, and indicates the fraction of the document displayed in the scroll window. Similarly, a window's title bar plays several roles: label, dragging handle, window activation indicator, and location for window control buttons. In the classic Mac interface, in fact, even the activation indicator played two roles. When the window was activated, closely spaced horizontal lines filled the title bar, giving it a perceived affordance for dragging.

11.7 Contrast & Visual Variables

- Contrast encodes information along visual dimensions



Contrast refers to perceivable differences along a visual dimension, such as size or color. Contrast is the irregularity in a design that communicates information or makes elements stand out. Simplicity says we should eliminate **unimportant** differences. Once we've decided that a difference is important, however, we should choose the dimension and degree of contrast in such a way that the difference is salient, easily perceptible, and appropriate to the task.

Crucial to this decision is an understanding of the different visual dimensions. Jacques Bertin developed a theory of visual variables that is particularly useful here (Bertin, Graphics and Graphics Information Processing, 1989). The seven visual variables identified by Bertin are shown above. Bertin called these dimensions retinal variables, in fact, because they can be compared effortlessly without additional cognitive processing, as if the retina were doing all the work. In terms of the model human processor abstraction, differences along these dimensions can be detected by the Perceptual Processor. Comparing numbers, on the other hand, would require the participation of the Cognitive Processor.

Each column in this display varies along only one of the seven variables. Most of the variables need no explanation, except perhaps for hue and value. **Hue** is pure color; **value** is the brightness or luminance of color. (Figure after Mullet & Sano, p. 54).

11.8 Characteristics of Visual Variables

- Scale = kinds of comparisons possible
  - Nominal (=)
    - All variables
  - Ordered (<, >)
    - Ordered: position, size, value, texture granularity
    - Not ordered: orientation, hue, shape
  - Quantitative (amount of difference)
    - Quantitative: position, size
    - Not quantitative: value, texture, orientation, hue, shape
- Length = number of distinguishable levels
  - Shape is very long (infinite variety)
  - Position is long and fine-grained
  - Orientation is very short (~ 4 levels)
  - Other variables are in between (~ 10 levels)

The visual variables are used for communication, by encoding data and drawing distinctions between visual elements. But the visual variables have different characteristics. Before you choose a visual variable to express some distinction, you should make sure that the visual variable's properties match your communication. For example, you could display a temperature using any of the dimensions: position on a scale, length of a bar, color of an indicator, or shape of an icon (a happy sun or a chilly icicle). Your choice of visual variable will strongly affect how your users will be able to perceive and use the displayed data.

Two characteristics of visual variables are the kind of scale and the length of the scale.

A **nominal** scale is just a list of categories. Only comparison for equality is supported by a nominal scale. Different values have no ordering relationship. The shape variable is purely nominal. Hue is also purely nominal, at least as a perceptual variable. Although the wavelength of light assigns an ordering to colors, the human perceptual system takes no notice of it. Likewise, there may be some cultural ordering imposed on hue (red is "hotter" than blue), but it's weak, doesn't relate all the hues, and is processed at a higher cognitive level.

An **ordered** scale adds an ordering to the values of the variable. Position, size, value, and to some extent texture (with respect to the grain size of the texture) are all ordered.

With a **quantitative** variable, you can perceive the amount of difference in the ordering. Position is quantitative. You can look at two points on a graph and tell that one is twice as high as the other. Size is also quantitative, but note that we are far better at perceiving quantitative differences in one dimension (i.e., length) than in two dimensions (area). Value is not quantitative; we can't easily perceive that one shade is twice as dark as another shade.

The **length** of a variable is the number of distinguishable values that can be perceived. We can recognize a nearly infinite variety of shapes, so the shape variable is very long, but purely nominal. Position is also long, and particularly fine-grained. Orientation, by contrast, is very short; only a handful of different orientations can be perceived in a display before confusion starts to set in. The other variables lie somewhere in between, with roughly 10 useful levels of distinction, although size and color are somewhat longer than value and texture.

11.9 Attention

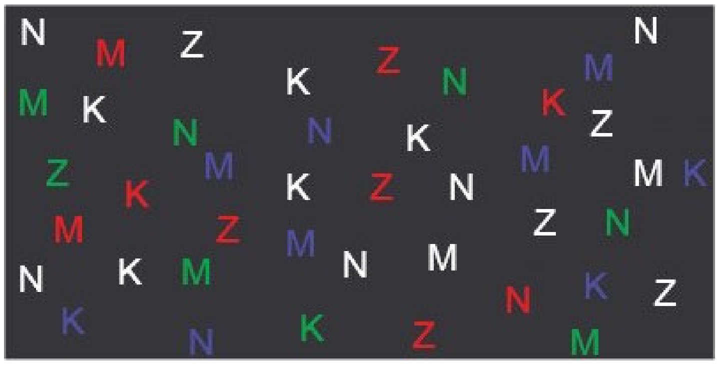
- Recall the spotlight metaphor
  - Attention spotlight moves serially from one input channel to another
  - All stimuli within spotlighted channel are processed in parallel
- Input channel = one or more visual variables
  - e.g., position, hue

11.10 Selectivity

- Selective perception: can attention be focused on one value of the variable, excluding other variables and values?
  - Selective: position, size, orientation, hue, value, texture
  - Not selective: shape

Selectivity is the degree to which a single value of the variable can be selected from the entire visual field. Most variables are selective: e.g., you can locate green objects at a glance, or tiny objects. Shape, however, is not selective in general. It's hard to pick out triangles amidst a sea of rectangles.

11.11



Ask yourself these questions:

- find all the letters on the left half of the page (**position**)
- find all the red letters (**hue**)
- find all the K's (**shape**)

Which of these questions felt easy to answer, and which felt hard? The easy ones were **selective** visual variables.

11.12 Associativity

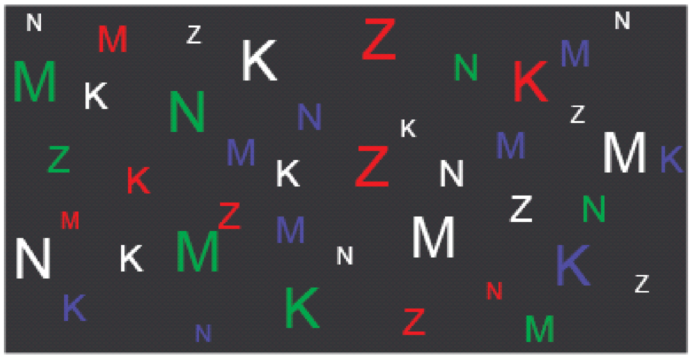
- Associative perception: can variable be ignored when looking at other variables?
  - Associative: position, hue, value, texture, shape, orientation

- Not associative: size, value
  - Small size and low value interfere with ability to perceive hue, value, texture, and shape

There are two ways that your choice of visual variables can affect the user’s ability to attend to them.

Associativity refers to how easy it is to ignore the variable, letting all of the distinctions along that dimension disappear. Variables with poor associativity interfere with the perception of other visual dimensions. In particular, size and value are dissociative, since tiny or faint objects are hard to make out.

11.13



Notice that when we use size as a visual variable as well, the shapes and hues of small objects become harder to detect.

11.14 Recall the Stroop Effect

Green
Orange
Red
Black
Pink
Blue

The Stroop effect, which we saw in an earlier lecture on human capabilities, was another kind of interference caused by our inability to selectively attend to colors or words. But the Stroop effect was really an interference at the **cognitive** level, not the **perceptual** level. You don’t read words in early stages of visual processing.

11.15 Techniques for Contrast

- Choose appropriate visual variables
- Use as much length as possible
- Sharpen distinctions for easier perception
  - Multiplicative scaling, not additive
  - Redundant coding where needed
  - Cartoonish exaggeration where needed
- Use the “squint test”

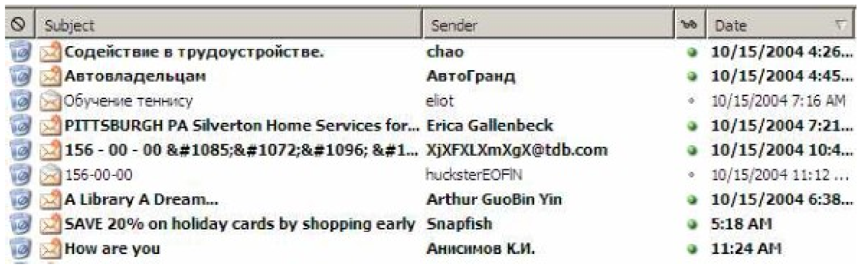
Once you've decided that a contrast is essential in your interface, **choose the right visual variable** to represent it, keeping in mind the data you're trying to communicate and the task users need to do with the data. For example, consider a text content hierarchy: title, chapter, section, body text, footnote. The data requires an ordered visual variable; a purely nominal variable like shape (e.g., font family) would not by itself be able to communicate the hierarchy ordering. If each element must communicate multiple independent dimensions of data at once (e.g., a graph that uses size, position, and color of points to encode different data variables), then you need to think about the effects of associativity and selectivity.

Once you've chosen a variable, use as much of the length of the variable as you can. Determine the minimum and maximum value you can use, and exploit the whole range. In the interests of simplicity, you should minimize the number of distinct values you use. But once you've settled on N levels, distribute those N levels as widely across the variable as is reasonable. For position, this means using the full width of the window; for size, it means using the smallest and the largest feasible sizes.

Choose variable values in such a way as to make sharp, easily perceptible distinctions between them. Multiplicative scaling (e.g., size growing by a factor of 1.5 or 2 at each successive level) is makes sharper distinctions than additive scaling (e.g., adding 5 pixels at each successive level). You can also use redundant coding, in several visual variables, to enhance important distinctions further. The title of a document is not only larger (size), but it's also centered (position), bold (value), and maybe a distinct color as well. Exaggerated differences can be useful, particularly when you're drawing icons: like a cartoonist, you have to give objects exaggerated proportions to make them easily recognizable.

The squint test is a technique that simulates early visual processing, so you can see whether the contrasts you've tried to establish are readily apparent. Close one eye and squint the other, to disrupt your focus. Whatever distinctions you can still make out will be visible "at a glance."

11.16 Choosing Visual Variables for a Display



Subject	Sender		Date
Содействие в трудоустройстве.	**chao**	●	**10/15/2004 4:26...**
Автовладельцам	**АвтоГранд**	●	**10/15/2004 4:45...**
Обучение теннису	eliot	•	10/15/2004 7:16 AM
PITTSBURGH PA Silverton Home Services for...	**Erica Gallenbeck**	●	**10/15/2004 7:21...**
156 - 00 - 00 &#1085;&#1072;&#1096; &#1...	**XjXFXLXmXgX@tdb.com**	●	**10/15/2004 10:4...**
156-00-00	hucksterEOFIN	•	10/15/2004 11:12 ...
A Library A Dream...	**Arthur GuoBin Yin**	●	**10/15/2004 6:38...**
SAVE 20% on holiday cards by shopping early	**Snapfish**	●	**5:18 AM**
How are you	**Анисимов К.И.**	●	**11:24 AM**

Let's look at an email inbox to see how data associated with email messages are encoded into visual variables in the display. Here are the data fields shown above, in columns from left to right:

Spam flag: nominal, 2 levels (spam or not)

Subject: nominal (but can be ordered alphabetically), infinite (but maybe only ~100 are active) **Sender**: nominal (but can be ordered alphabetically), infinite (but maybe ~100 people you know + everybody else are useful simplifications)

Unread flag: nominal, 2 levels (read or unread)

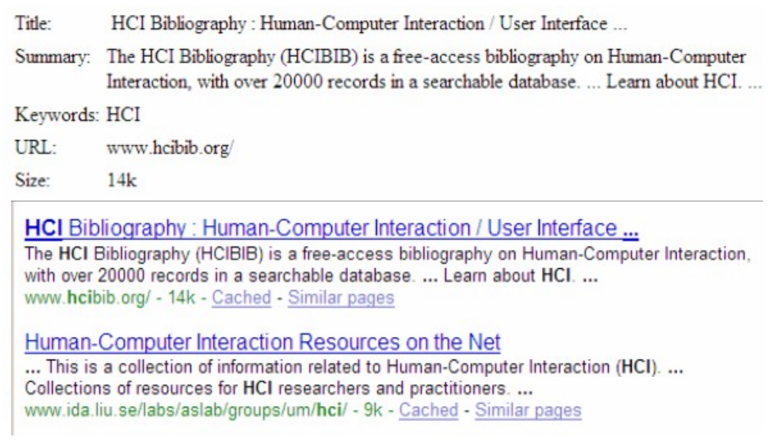
Date: quantitative (but maybe ordered is all that matters), infinite (but maybe only ~10 levels matter: today, this week, this month, this year, older)

This information is **redundantly** coded into visual variables in the display shown above, for better contrast. First, all the fields use position as a variable, since each is assigned to a different column. In addition:

Spam: shape, hue, value, size (big colorful icon vs. little dot)
Subject: shape
Sender: shape
Unread: shape, hue, value, size (big green dot vs. little gray dot) and value of entire line (boldface vs. non)
Date: shape, size (today is shorter than earlier dates), position (list is sorted by date)
Exercise: try designing a visualization with these encodings instead:

Spam: size (this takes advantage of dissociativity)
Subject: shape Sender: position
Unread: value
Date: position

11.17 Designing Information Displays



Title: HCI Bibliography : Human-Computer Interaction / User Interface ...
Summary: The HCI Bibliography (HCIBIB) is a free-access bibliography on Human-Computer Interaction, with over 20000 records in a searchable database. ... Learn about HCI. ...
Keywords: HCI
URL: www.hcibib.org/
Size: 14k

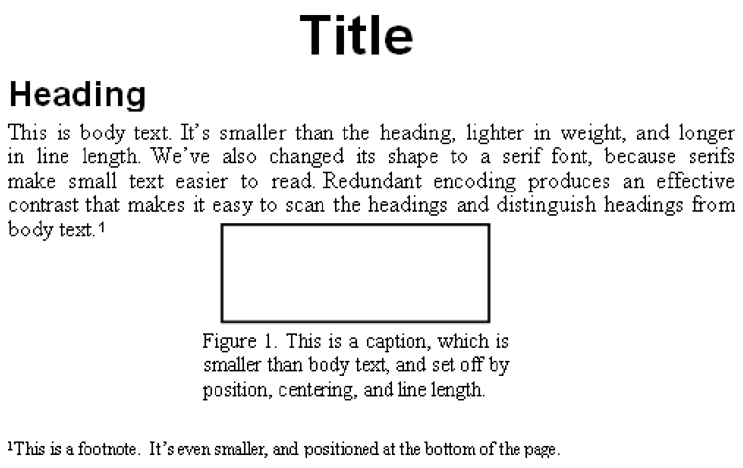
HCI Bibliography : Human-Computer Interaction / User Interface **...**
The **HCI** Bibliography (HCIBIB) is a free-access bibliography on Human-Computer Interaction, with over 20000 records in a searchable database. **...** Learn about **HCI**. **...**
www.**hci**bib.org/ - 14k - Cached - Similar pages

Human-Computer Interaction Resources on the Net
... This is a collection of information related to Human-Computer Interaction (**HCI**). **...** Collections of resources for **HCI** researchers and practitioners. **...**
www.ida.liu.se/labs/aslab/groups/um/**hci**/ - 9k - Cached - Similar pages

Here's another example showing how redundant encoding can make an information display easier to scan and easier to use. Search engine results are basically just database records, but they aren't rendered in a simplistic caption/field display like the one shown on top. Instead, they use rich visual variables – and no field labels! – to enhance the contrast among the items. Page titles convey the most information, so they use size, hue, and value (brightness), plus a little shape (the underline). The summary is in black for good readability, and the URL and size are in green to bracket the summary.

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

11.18 Contrast in Publication styles



Title

Heading

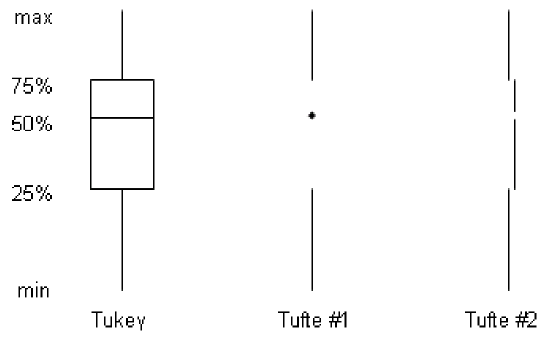
This is body text. It's smaller than the heading, lighter in weight, and longer in line length. We've also changed its shape to a serif font, because serifs make small text easier to read. Redundant encoding produces an effective contrast that makes it easy to scan the headings and distinguish headings from body text.¹

Figure 1. This is a caption, which is smaller than body text, and set off by position, centering, and line length.

¹This is a footnote. It's even smaller, and positioned at the bottom of the page.

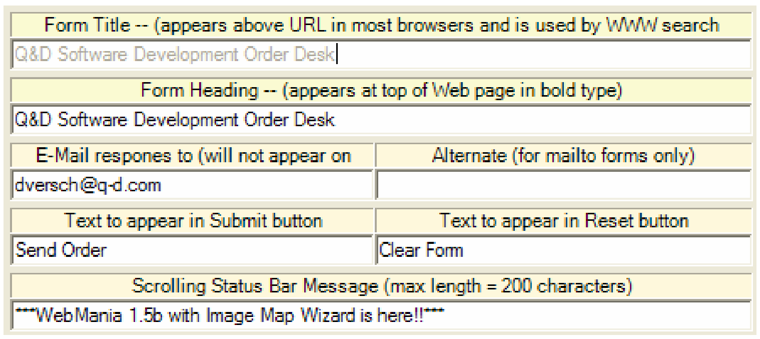
Titles, headings, body text, figure captions, and footnotes show how contrast is used to make articles easier to read. You can do this yourself when you're writing papers and documentation. Does this mean contrast should be maximized by using lots of different fonts like Gothic and Bookman? No, for two reasons – contrast must be balanced against simplicity, and text shape variations aren't the best way to establish contrast.

11.19 Simplicity vs. Contrast



Conversely, here's a case where simplicity is taken too far, and contrast suffers. Simplicity and contrast seem to fight with each other. The standard Tukey box plot shows 5 different statistics in a single figure. But it has unnecessary lines in it! Following the principle of simplicity to its logical extreme, Edward Tufte proposed two simplifications of the box plot which convey exactly the same information – but at a great cost in contrast. Try the squint test on the Tukey plot, and on Tufte's second design. What do you see?

11.20 Contrast Problems



Form Title -- (appears above URL in most browsers and is used by WWW search

Q&D Software Development Order Desk

Form Heading -- (appears at top of Web page in bold type)

Q&D Software Development Order Desk

E-Mail respones to (will not appear on	Alternate (for mailto forms only)
dversch@q-d.com	
Text to appear in Submit button	Text to appear in Reset button
Send Order	Clear Form

Scrolling Status Bar Message (max length = 200 characters)

WebMania 1.5b with Image Map Wizard is here!!

Here's an example of too little contrast. It's important to distinguish captions from text fields, but in this design, most of the visual variables are the same for both:

- the **position** is very similar: the box around each caption and text field begins at the same horizontal position. The text itself begins at different positions (left-justified vs. aligned), but it isn't a strong distinction, and some of the captions fill their column.
- the **size** is the same: captions and text fields fill the same column width
- the background **hue** is slightly different (yellow vs. white), but not easily differentiable by the squint test
- the background **value** is the same (very bright)
- the foreground **hue** and **value** are the same (black, plain font)
- the **orientation** is the horizontal, because of course you have to read it.

The result is that it's hard to scan this form. The form is also terribly crowded, which leads us into our next topic…

11.21 White Space

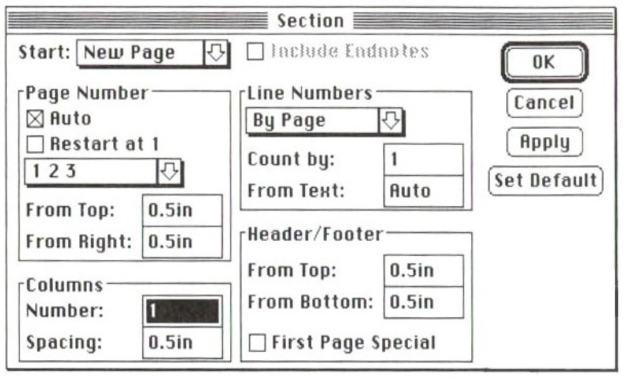
- Use white space for grouping, instead of lines
- Use margins to draw eye around design
- Integrate figure and ground
  - Object should be scaled proportionally to its background
- Don′t crowd controls together
  - Crowding creates spatial tension and inhibits scanning

White space plays an essential role in composition. Screen real estate is at a premium in many graphical user interfaces, so it’s a constant struggle to balance the need for white space against a desire to pack information and controls into a display. But insufficient white space can have serious side-effects, making a display more painful to look at and much slower to scan.

Put margins around all your content. Labels and controls that pack tightly against the edge of a window are much slower to scan. When an object is surrounded by white space, keep a sense of proportion between the object (the figure) and its surroundings (ground). Don’t crowd controls together, even if you’re grouping the controls.

Crowding inhibits scanning, and produces distracting effects when two lines (such as the edges of text fields) are too close. Many UI toolkits unfortunately encourage this crowding by packing controls tightly together by default, but Java Swing (at least) lets you add empty margins to your controls that give them a chance to breathe.

11.22 Crowded Dialog



Start: New Page
☐ Include Endnotes

Page Number
☒ Auto
☐ Restart at 1
1 2 3
From Top: 0.5in
From Right: 0.5in

Columns
Number: 1
Spacing: 0.5in

Line Numbers
By Page
Count by: 1
From Text: Auto

Header/Footer
From Top: 0.5in
From Bottom: 0.5in
☐ First Page Special

Here’s an example of an overcrowded dialog. The dialog has no margins around the edges; the controls are tightly packed together; and lines are used for grouping where white space would be more appropriate. Screen real estate isn’t terribly precious in a transient dialog box.

The crowding leads to some bad perceptual effects. Lines appearing too close together – such as the bottom of the Spacing text field and the group line that surround it – blend together into a thicker, darker line, making a wart in the design. A few pixels of white space between the lines would completely eliminate this problem.

11.24 Using White Space to Set Off Lebels

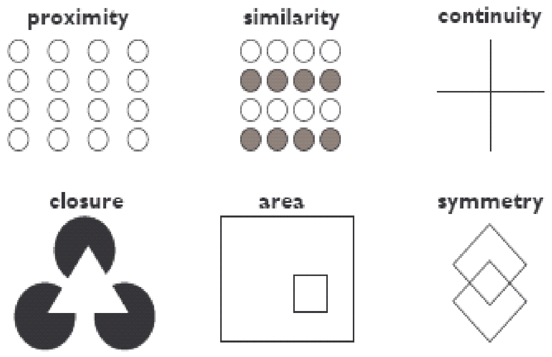
Images found in: K. Mullet and D. Sano, "Designing visual interfaces: Communication-oriented techniques,“ Sunsoft Press, Prentice Hall (1995), p. 96.

A particularly effective use of white space is to put labels in the left margin, where the white space sets off and highlights them. In dialog box (a), you can't scan the labels and group names separately; they interfere with each other, as do the grouping lines. In the redesigned dialog (b), the labels are now alone on the left, making them much easier to scan.

For the same reason, you should put labels to the left of controls, rather than above.

11.25 The Gestalt Principles of Grouping

- Gestalt principles explain how eye creates a whole (gestalt) from parts



The power of white space for grouping derives from the Gestalt principle of proximity. These principles, discovered in the 1920's by the Gestalt school of psychologists, describe how early visual processing groups elements in the visual field into larger wholes. Here are the six principles identified by the Gestalt psychologists:

Proximity. Elements that are closer to each other are more likely to be grouped together. You see four vertical columns of circles, because the circles are closer vertically than they are horizontally.

Similarity. Elements with similar attributes are more likely to be grouped. You see four *rows* of circles in the Similarity example, because the circles are more alike horizontally than they are vertically.

Continuity. The eye expects to see a contour as a continuous object. You primarily perceive the Continuity example above as two crossing lines, rather than as four lines meeting at a point, or two right angles sharing a vertex.

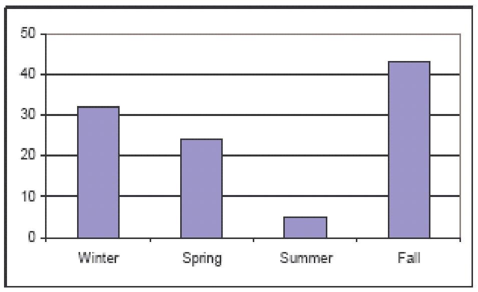
Closure. The eye tends to perceive complete, closed figures, even when lines are missing. We see a triangle in the center of the Closure example, even though its edges aren't complete.

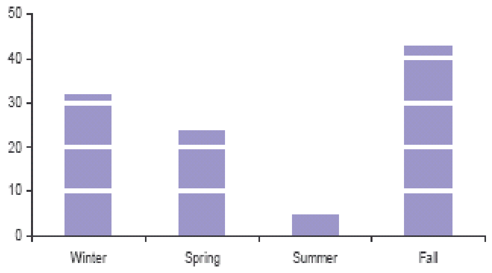
Area. When two elements overlap, the smaller one will be interpreted as a figure in

front of the larger ground. So we tend to perceive the Area example as a small square in front of a large square, rather than a large square with a hole cut in it.

Symmetry. The eye prefers explanations with greater symmetry. So the Symmetry example is perceived as two overlapping squares, rather than three separate polygons.

11.26 White Space Avoids Visual Noise

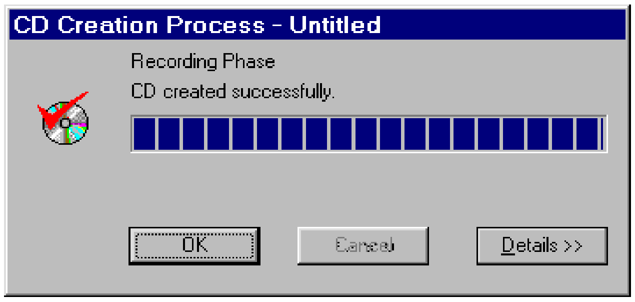


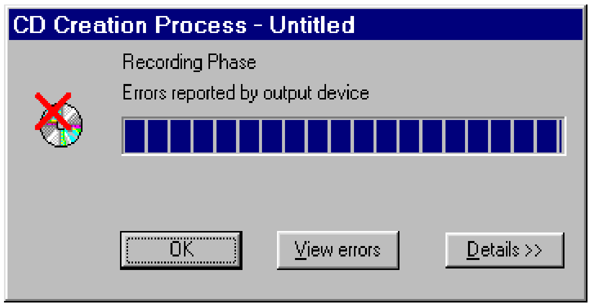


Here's an interesting idea from Tufte: get rid of the grid rules on a standard bar chart, and use whitespace to show where the grid lines would cross the bars. It's much less noisy. (But alas, impossible to do automatically in Excel.).

Lecture 12: Computer Prototyping

12.1 UI Hall of Fame or Shame?





CD Creation Process - Untitled
Recording Phase
Errors reported by output device
OK
View errors
Details >>

http://images.google.com/images?q=color+easycd2.gif
http://images.google.com/images?q=color+easycd1.gif

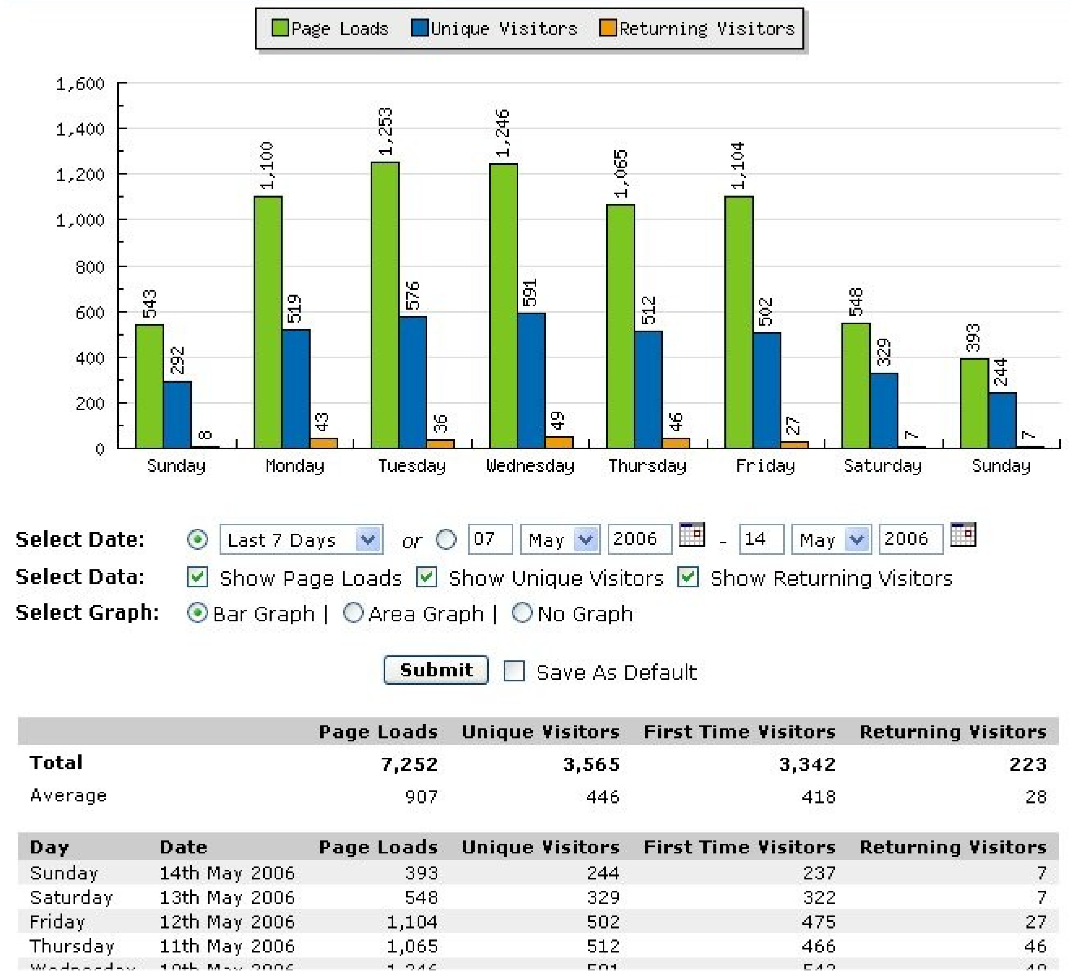
Our Hall of Shame candidate for the day is this dialog box from Adaptec Easy CD Creator, which appears at the end of burning a CD. The top image shows the dialog when the CD was burned successfully; the bottom image shows what it looks like when there was an error.

The key problem is the **lack of contrast** between these two states. Success or failure of CD burning is important enough to the user that it should be obvious at a glance. But these two dialogs look identical at a glance. How can we tell? Use the **squint test**, which we talked about in the graphic design lecture. When you're squinting, you see some labels, a big filled progress bar, a roundish icon with a blob of red, and three buttons. All the details, particularly the text of the messages and the exact shapes of the red blob, are fuzzed out. This simulates what a user would see at a quick glance, and it shows that the graphic design doesn't convey the contrast.

One improvement would change the check mark to another color, say green or black. Using red for OK seems **inconsistent** with the real world, anyway. But designs that differ only in red and green wouldn't pass the squint test for color-blind users.

Another improvement might remove the completed progress bar from the error dialog, perhaps replacing it with a big white text box containing a more detailed description of the problem. That would clearly pass the squint test, and make errors much more noticeable.

12.2 Hall of Fame



Select Date: [✓ Last 7 Days] or 07 May 2006 - 14 May 2006

Select Data: ✓ Show Page Loads ✓ Show Unique Visitors ✓ Show Returning Visitors

Select Graph: [✓ Bar Graph] | Area Graph | No Graph

Submit ☐ Save As Default

	Page Loads	Unique Visitors	First Time Visitors	Returning Visitors
Total	**7,252**	**3,565**	**3,342**	**223**
Average	907	446	418	28

Day	Date	Page Loads	Unique Visitors	First Time Visitors	Returning Visitors
Sunday	14th May 2006	393	244	237	7
Saturday	13th May 2006	548	329	322	7
Friday	12th May 2006	1,104	502	475	27
Thursday	11th May 2006	1,065	512	466	46

http://my.statcounter.com/project/standard/stats.php?project_id=234043&guest=1

Here’s our hall of fame example. StatCounter is a web site that tracks usage statistics of your web site, using a hit counter. This is a sample of its statistics page.

The first thing to note is the simplicity of the design. Only a few colors are used: gray, black, and a few shades of blue, predominately. This simplicity allows the few different colors – like the red TIP

– to really stand out. The design also omits unnecessary labels – for example, the date entry boxes on the bottom don’t need labels like “From:” and “To:”, because they’re self-describing.

It’s interesting to look at how visual variables were used to encode the information. Position is used to represent date (horizontally) and number of hits (vertically). The kind of statistic is encoded in the value of the graph line, ranging from dark blue (returning visitors) to light blue (page loads). The statistics are actually related in a hierarchy: since every returning visitor is a unique visitor, and every unique visitor causes at least one page load, it is always the case that page loads > unique visitors > returning visitors. This hierarchy is emphasized both by position (the curves are always in the same order vertically) and by value. Position and value were good choices for emphasizing the ordering, because both variables are ordered. An unordered visual variable, like the shape of the data point, would not have been as effective.

This page does have some problems. One is the use of two different terms, “range” and “period”, which basically mean the same thing (internal consistency). The Set Period interface is in fact a list of common shortcuts, like “the last 30 days”, which is a good thing; but the shortcuts should be presented more prominently. There’s no reason why the year field (2004) should be a text box, rather than a drop-down with choices appropriate to the actual range of data available (error prevention). And the hyphen between the start date and the end date is too small to have good contrast with the controls around it; it disappears.

12.3 Today's Topics

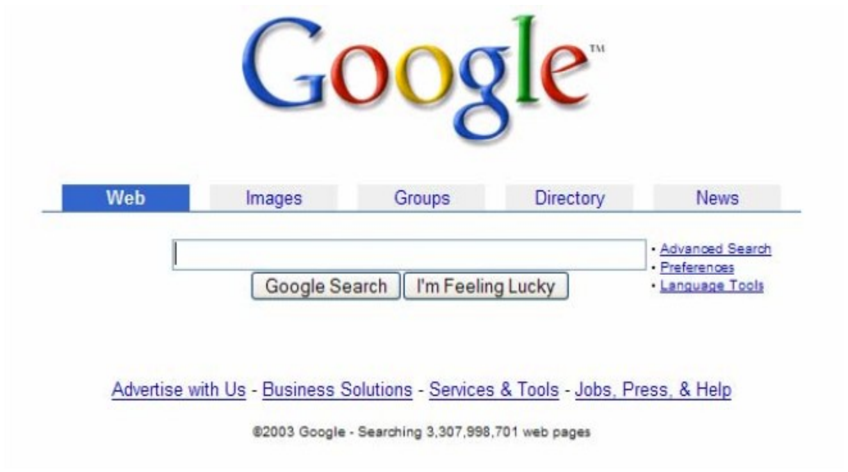
- Balance
- Alignment
- Color guidelines
- Computer prototyping

12.4 Balance & Symmetry

- Choose an axis (usually vertical)
- Distribute elements equally around the axis
  - Equalize both mass and extent

Balance and symmetry are valuable tools in a designer's toolkit. In graphic design, symmetry rarely means exact, mirror-image equivalence. Instead, what we mean by symmetry is more like balance: is there the same amount of stuff on each side of the axis of symmetry. We measure "stuff" by both mass (quantity of nonwhite pixels) and extent (area covered by those pixels); both mass and extent should be balanced.

12.5 Symmetry Example

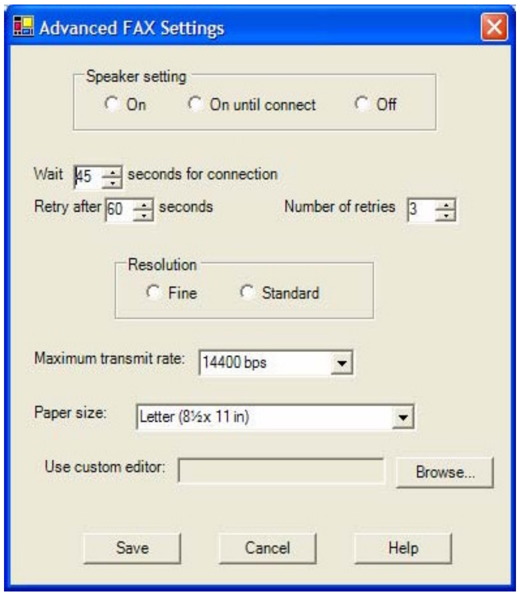

Google™
Web
Images
Groups
Directory
News
• Advanced Search
• Preferences
• Language Tools
Google Search
I'm Feeling Lucky
Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help
©2003 Google - Searching 3,307,998,701 web pages

An easy way to achieve balance is to simply center the elements of your display. That automatically achieves balance around a vertical axis. If you look at Google's home page, you'll see this kind of approach in action. In fact, only one element of the Google home page breaks this symmetry: the stack of links for Advanced Search, Preferences, and Language Tools on the right. This slight irregularity actually helps emphasize these links slightly.

Symmetry is a kind of **simplicity**; asymmetry creates a **contrast**. Use that contrast wisely.

12.6 Alignment

- Align labels on left or right
- Align controls on left and right
  - Expand as needed
- Align text baselines


Advanced FAX Settings

Speaker setting
( ) On ( ) On until connect ( ) Off

Wait 45 seconds for connection
Retry after 60 seconds Number of retries 3

Resolution
( ) Fine ( ) Standard

Maximum transmit rate: 14400 bps

Paper size: Letter (8½x 11 in)

Use custom editor: Browse...

Save Cancel Help

Finally, simplify your designs by aligning elements horizontally and vertically. Alignment contributes to the simplicity of a design. Fewer alignment positions means a simpler design. The dialog box shown has totally haphazard alignment, which makes it seem more complicated than it really is.

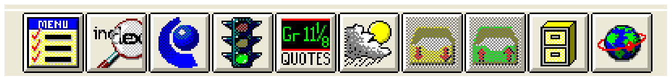
Labels (e.g. "Wait" and "Retry after"). There are two schools of thought about label alignment: one school says that the left edges of labels should be aligned, and the other school says that their right edges (i.e., the colon following each label) should be aligned. Both approaches work, and experimental studies haven't found any significant differences between them. Both approaches also fail when long labels and short labels are used in the same display. You'll get best results if you can make all your labels about the same size, or else break long labels into multiple lines.

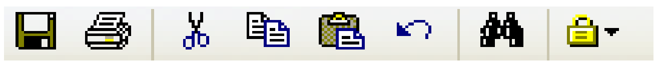
Controls (e.g., text fields, combo boxes, checkboxes). A column of controls should be aligned on both the left and the right. Sometimes this seems unreasonable -- should a short date field be expanded to the same length as a filename? It doesn't hurt the date to be larger than necessary, except perhaps for reducing its perceived affordance for receiving a date. You can also solve these kinds of problems by rearranging the display, moving the date elsewhere, although be careful of disrupting your design's functional grouping or the expectations of your user.

So far we've only discussed left-to-right alignment. Vertically, you should ensure that labels and controls on the same row share the same text baseline. Java Swing components are designed so that text baselines are aligned if the components are centered vertically with respect to each other, but not if the components' tops or bottoms are aligned. Java AWT components are virtually impossible to align on their baselines. The dialog shown here has baseline alignment problems, particularly between the combo boxes and their captions.

12.7 Color Guidelines

- Remember limitations of human vision
  - Color blindness, red-on-blue, small blue details
- Use few colors
- Avoid saturated colors
- Be consistent and match expectations





We've already talked a lot about the limits of human color vision. In general, colors should be used sparingly. An interface with many colors appears more complex, more cluttered, and more distracting. Use only a handful of colors.

Background colors should establish a good contrast with the foreground. White is a good choice, since it provides the most contrast; but it also produces bright displays, since our computer displays emit light rather than reflecting it. Pale (desaturated) yellow and very light gray are also good background colors.

In general, avoid strongly saturated colors – i.e., the colors around the top edge of the HSV cone. Saturated colors can cause visual fatigue because the eye must keep refocusing on different wavelengths. They also tend to saturate the viewer's receptors (hence the name). One study found that air traffic controllers who viewed strongly saturated green text on their ATC interfaces for many hours had trouble seeing pink or red (the other end of the red/green color channel) for up to 15 minutes after their shift was over.

Use less saturated versions instead, pushing them towards gray.

To sharpen contrasts, you can use opponent colors: red/green, blue/yellow. But keep color blind users in mind; hue should not be the only way you establish the contrast. Both color-blind and color- normal users will see the contrast better if you vary both hue and value.

Finally, match expectations. One of the problems with the Adaptec dialogs at the beginning of this lecture was the use of red for OK. Red generally means stop, warning, error, or hot. Green conventionally means go, or OK. Yellow means caution, or slow.

12.8 Paper Prototyping is Not Enough

- Low fidelity in:
  - Look
  - Feel
  - Dynamics
  - Response time
  - Context
- Users can't try it without a human to simulate computer

We now turn to prototyping. Paper prototyping is neat, but it's not enough. We discussed some of these drawbacks in the paper prototyping lecture.

First, paper prototypes are low-fi in look. It's sometimes hard for users to recognize widgets that you've hand-drawn, or labels that you've hastily scribbled. A paper prototype won't tell you what will actually fit on the screen, since your handwritten font and larger-than-life posterboard aren't realistic simulations. A paper prototype can't easily simulate some important characteristics of your interface's look – for example, does the selection highlighting have enough contrast, using visual variables that are selective, does it float above the rest of the design in its own layer? You have to make decisions about graphic design at some point, and you want to iterate those decisions just like other aspects of usability. So we need another prototyping method. Paper prototypes are also low-fi in feel. Finger & pen doesn't behave like mouse & keyboard (e.g., the drag & drop issue mentioned earlier). Even pen-based computers don't really feel like pen & paper. You can't test Fitts's Law issues, like whether a button is big enough to click on.

Paper is low-fi in dynamic feedback. You don't get any animation, like spinning globes, moving progress bars, etc. You can't test mouse-over feedback, as we mentioned earlier.

It's also low-fi in response time, because the human computer is far slower than the real computer. So you can't measure how long it takes users to do a task with your interface.

Finally, paper is low-fi with respect to context of use. A paper prototype can only really be used on a tabletop, in office-like environment. Users can't take it to grocery store, subway, aircraft carrier deck, or wherever the target interface will actually be used. If it's a handheld application, the ergonomics are all wrong; you aren't holding it in your hand, and it's not the right weight.

12.9 Computer Prototype

- Interactive software simulation
- High-fidelity in look & feel
- Low-fidelity in depth
  - Paper prototype had a human simulating the backend; computer prototype doesn't
  - Computer prototype is typically **horizontal**: covers most features, but no backend

So at some point we have to depart from paper and move our prototypes into software. A typical computer prototype is a **horizontal** prototype. It's high-fi in look and feel, but low-fi in depth – there's no backend behind it. Where a human being simulating a paper prototype can generate new content on the fly in response to unexpected user actions, a computer prototype cannot.

12.10 What You Can Learn From Computer Prototypes

- Everything you learn from a paper prototype, plus:
- Screen layout
  - Is it clear, overwhelming, distracting, complicated?
  - Can users find important elements?
- Colors, fonts, icons, other elements
  - Well-chosen?
- Interactive feedback
  - Do users notice & respond to status bar messages, cursor changes, other feedback
- Fitts's Law issues
  - Controls big enough? Too close together? Scrolling list is too long?

Computer prototypes help us get a handle on the graphic design and dynamic feedback of the interface.

12.11 Why Use Prototyping Tools?

- Faster than coding
- No debugging
- Easier to change or throw away
- Don't let Java do your graphic design

One way to build a computer prototype is just to program it directly in an implementation language, like Java or C++, using a user interface toolkit, like Swing or MFC. If you don't hook in a backend, or use stubs instead of your real backend, then you've got a horizontal prototype.

But it's often better to use a prototyping tool instead. Building an interface with a tool is usually faster than direct coding, and there's no code to debug. It's easier to change it, or even throw it away if your design turns out to be wrong. Recall Cooper's concerns about prototyping: your computer prototype may become so elaborate and precious that it becomes your final implementation, even though (from a software engineering point of view) it might be sloppily designed and unmaintainable.

Also, when you go directly from paper prototype to code, there's a tendency to let your UI toolkit handle all the graphic design for you. That's a mistake. For example, Java has layout managers that automatically arrange the components of an interface. Layout managers are powerful tools, but they produce horrible interfaces when casually or lazily used. A prototyping tool will help you envision your interface and get its graphic design right first, so that later when you move to code, you know what you're trying to persuade the layout manager to produce.

Even with a prototyping tool, computer prototypes can still be a tremendous amount of work. When drag & drop was being considered for Microsoft Excel, a couple of Microsoft summer interns were assigned to develop a prototype of the feature using Visual Basic. They found that they had to implement a substantial amount of basic spreadsheet functionality just to test drag & drop. It took two interns their entire summer to build the prototype that proved that drag & drop was useful. Actually adding the feature to Excel took a staff programmer only a week. This isn't a fair comparison, of course – maybe six intern-months was a cost worth paying to mitigate the risk of one fulltimer-week, and the interns certainly learned a lot. But building a computer prototype can be a slippery slope, so don't let it suck you in too deeply. Focus on what you want to test, i.e., the design risk you need to mitigate, and only prototype that.

12.12 Prototyping Techniques

- Storyboard
  - Sequence of painted screenshots connected by hyperlinks ("hotspots")
- Form builder
  - Real windows assembled from a palette of widgets (buttons, text fields, labels, etc.)

There are two major techniques for building a computer prototype.

A **storyboard** is a sequence (a graph, really) of fixed screens. Each screen has one or more **hotspots** that you can click on to jump to another screen. Sometimes the transitions between screens also involve some animation in order to show a dynamic effect, like mouse-over feedback or drag-drop feedback.

A **form builder** is a tool for drawing real, working interfaces by dragging widgets from a palette and positioning them on a window.

12.13 Storyboarding Tools

- HTML
  - image maps
- Flash/Director
  - animation + actions
- PowerPoint
  - images + links + animation
- All these tools have scripting languages, too
  - Help orchestrate the transitions
- For high fidelity look, take screenshots of widgets from a form builder

Here are some tools commonly used for storyboarding.

A PowerPoint presentation is just a slide show. Each slide shows a fixed screenshot, which you can draw in a paint program and import, or which you can draw directly in PowerPoint. A PowerPoint storyboard doesn't have to be linear slide show. You can create hyperlinks that jump to any slide in the presentation.

Macromedia Flash (formerly Director) is a tool for constructing multimedia interfaces. It's particularly useful for prototyping interfaces with rich animated feedback.

HTML is also useful for storyboarding. Each screen is an imagemap. Macromedia Dreamweaver makes it easy to build HTML imagemaps.

All these tools have scripting languages – PowerPoint has Basic, Flash/Director has a language called Lingo, and HTML has Javascript – so you can write some code to orchestrate transitions, if need be.

If your storyboards need standard widgets like buttons or text boxes, you can create some widgets in a form builder and take static screenshots of them for your storyboard.

You can find Flash, Director, and Dreamweaver installed in MIT's New Media Center (search for it in Google), a cluster of Macs on the first floor of building 26. The room is sometimes used for classes during the day, but is open to the MIT community at other times.

12.14 Pros & Cons of Storyboarding

- Pros
  - You can draw anything
- Cons
  - No text entry
  - Widgets aren't active
  - "Hunt for the hotspot"

The big advantage of storyboarding is similar to the advantage of paper: you can draw anything on a storyboard. That frees your creativity in ways that a form builder can't, with its fixed palette of widgets.

The disadvantages come from the storyboard's static nature. All you can do is click, not enter text. You can still have text boxes, but clicking on a text box might make its content magically appear, without the user needing to type anything. Similarly, scrollbars, list boxes, and buttons are just pictures, not active widgets. Watching a real user in front of a storyboard often devolves into a game of "hunt for the hotspot", like children's software where the only point is to find things on the screen to click on and see what they do. The hunt-for-the-hotspot effect means that storyboards are largely useless for user testing, unlike paper prototypes. In general, horizontal computer prototypes are better evaluated with other techniques, like heuristic evaluation.

12.15 Form Builders

- HTML pages and forms
  - Natural if you're building a web application
  - May have low-fidelity look otherwise
- Java GUI builders
  - Sun NetBeans
  - Eclipse Visual Editor
  - Borland JBuilder
- Other GUI builders
  - Visual Basic, .NET Windows Forms
  - Mac Interface Builder
  - Qt Designer
- Tips
  - Use absolute positioning for now

Here are some form builder tools.

HTML is a natural tool to use if you're building a web application. You can compose static HTML pages simulating the dynamic responses of your web interface. Although the responses are canned, your prototype is still better than a storyboard, because its screens are more active than mere screenshots: the user can actually type into form fields, scroll through long displays, and see mouse- over feedback. Even if you're building a desktop or handheld application, HTML may still be useful. For example, you can mix static screenshots of some parts of your UI with HTML form widgets (buttons, list boxes, etc) representing the widget parts. It may be hard to persuade HTML to render a desktop interface in a high-fidelity way, however.

Visual Basic is the classic form builder. Many custom commercial applications are built entirely with Visual Basic.

There are several form builders for Java. Sun NetBeans and Borland JBuilder (Personal Edition) can be downloaded free.

Be careful when you're using a form builder for prototyping to avoid layout managers when you're doing your initial graphic designs. Instead, use absolute positioning, so you can put each component where you want it to go. Java GUI builders may need to be told not to use a layout manager.

12.16 Pros & Cons of Form Builders

- Pros
  - Actual controls, not just pictures of them
  - Can hook in some backend if you need it
    - But then you won't want to throw it away
- Cons
  - Limits thinking to standard widgets
  - Useless for rich graphical interfaces

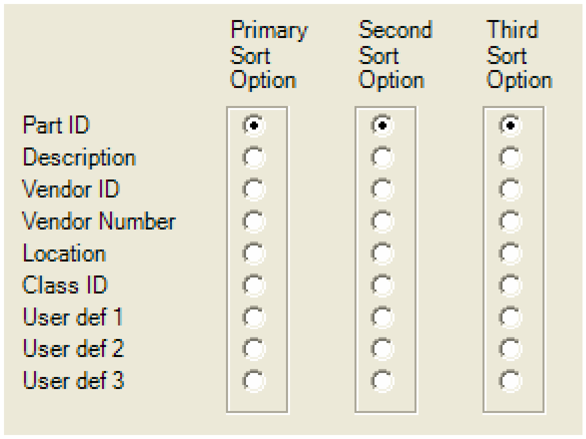
Unlike storyboards, form builders use actual working widgets, not just static pictures. So the widgets look the same as they will in the final implementation (assuming you're using a compatible form builder – a prototype in Visual Basic may not look like a final implementation in Java).

Also, since form builders usually have an implementation language underneath them – which may even be the same implementation language that you'll eventually use for your final interface -- you can also hook in as much or as little backend as you want.

On the down side, form builders give you a fixed palette of standard widgets, which limits your creativity as a designer, and which makes form builders largely useless for prototyping rich graphical interfaces, e.g., a circuit-drawing editor. Form builders are great for the menus and widgets that surround a graphical interface, but can't simulate the "insides" of the application window.

Lecture 13: Toolkits

13.1 UI Hall of Fame or Shame?



	Primary Sort Option	Second Sort Option	Third Sort Option
Part ID	●	●	●
Description	○	○	○
Vendor ID	○	○	○
Vendor Number	○	○	○
Location	○	○	○
Class ID	○	○	○
User def 1	○	○	○
User def 2	○	○	○
User def 3	○	○	○

Our Hall of Shame candidate for the day is this interface for choosing how a list of database records should be sorted. Like other sorting interfaces, this one allows the user to select more than one field to sort on, so that if two records are equal on the primary sort key, they are next compared by the secondary sort key, and so on.

On the plus side, this interface communicates one aspect of its model very well. Each column is a set of radio buttons, clearly grouped together (both by **gestalt proximity** and by an explicit raised border). Radio buttons have the property that only one can be selected at a time. So the interface has a clear **affordance** for picking only one field for each sort key.

But, on the down side, the radio buttons don't afford making NO choice. What if I want to sort by only one key? I have to resort to a trick, like setting all three sort keys to the same field. The interface model clearly doesn't map correctly to the task it's intended to perform. In fact, unlike typical model mismatch problems, both the user and the system have to adjust to this silly interface model – the user by selecting the same field more than once, and the system by detecting redundant selections in order to avoid doing unnecessary sorts.

The interface also fails on **minimalist** design grounds. It wastes a huge amount of screen real estate on a two-dimensional matrix, which doesn't convey enough information to merit the cost. The column labels are similarly redundant; "sort option" could be factored out to a higher level heading.

13.2 Today's Topics

- Widgets
- Toolkit layering
- Look-and-feel

Today we'll finish up our survey of user interface implementation.

Last lecture, we discussed the component model, stroke model, and pixel model, and observed that virtually every graphical user interface uses all three of these models for output. The main decision is, at what point in a GUI application do you make the switch from one model to the next. The switch from component model to stroke model occurs at the leaves of the view hierarchy (although parent containers may also draw strokes, of course). The switch from stroke model to pixel model usually occurs within the system's graphics library.

13.3 Widgets

- Reusable user interface components
  - Also called controls, interactors, gizmos, gadgets
- Examples
  - Buttons, checkboxes, radio buttons
  - List boxes, combo boxes, drop-downs
  - Menus, toolbars
  - Scrollbars, splitters, zoomers
  - One-line text, multiline text, rich text
  - Trees, tables
  - Simple dialogs

Widgets are the last part of user interface toolkits we'll look at. Widgets are a success story for user interface software, and for object-oriented programming in general. Many GUI applications derive substantial reuse from widgets in a toolkit.

13.4 Widget Pros and Cons

- Advantages
  - Reuse of development effort
    - Coding, testing, debugging, maintenance
    - Iteration and evaluation
  - External consistency
- Disadvantages
  - Constrain designer's thinking
  - Encourage menu & forms style, rather than richer direct manipulation style
  - May be used inappropriately

Widget reuse is beneficial in two ways, actually. First are the conventional software engineering benefits of reusing code, like shorter development time and greater reliability. A widget encapsulates a lot of effort that somebody else has already put in.

Second are usability benefits. Widget reuse increases consistency among the applications on a platform. It also (potentially) represents usability effort that its designers have put into it. A scrollbar's affordances and behavior have been carefully designed, and hopefully evaluated. By reusing the scrollbar widget, you don't have to do that work yourself.

One problem with widgets is that they constrain your thinking. If you try to design an interface using a GUI builder – with a palette limited to standard widgets – you may produce a clunkier, more complex interface than you would if you sat down with paper and pencil and allowed yourself to think freely. A related problem is that most widget sets consist mostly of form-style widgets: text fields, labels, checkboxes – which leads a designer to think in terms of menu/form style interfaces. There are few widgets that support direct visual representations of application objects, because those representations are so application-dependent. So if you think too much in terms of widgets, you may miss the possibilities of direct manipulation.

Finally, widgets can be abused, applied to UI problems for which they aren't suited. We saw an example in Lecture 1 where a scrollbar was used for selection, rather than scrolling.

13.5 Widget Design

- Widget is a view + controller
  - Embedded model
    - Application data must be copied into the widget
    - Changes must be copied out'
  - Linked model

• Application provides model satisfying an interface
• Enables "data-bound" widgets, e.g. a table showing thousands of database rows, or a combo box with thousands of choices

Widgets generally combine a view and a controller into a single tightly-coupled object. For the widget's model, however, there are two common approaches. One is to fuse the model into the widget as well, making it a little MVC complex. With this embedded model approach, application data must be copied into the widget to initialize it. When the user interacts with the widget, the user's changes or selections must be copied back out.

The other alternative is to leave the model separate from the widget, with a well-defined interface that the application can implement.

Embedded models are usually easier for the developer to understand and use for simple interfaces, but suffer from serious scaling problems. For example, suppose you want to use a table widget to show the contents of a database. If the table widget had an embedded model, you would have to fetch the entire database and load it into the table widget, which may be prohibitively slow and memory-intensive. Furthermore, most of this is wasted work, since the user can only see a few rows of the table at a time. With a well-designed linked model, the table widget will only request as much of the data as it needs to display.

The linked model idea is also called **data binding**.

13.6 Toolkits

- User interface toolkit consists of:
  - Components (view hierarchy)
  - Stroke drawing
  - Pixel model
  - Input handling
  - Widgets

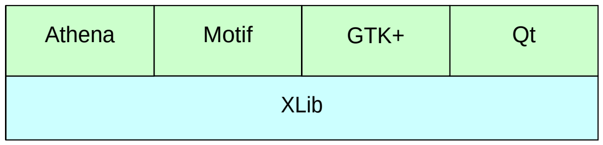
By now, we've looked at all the basic pieces of a user interface toolkit: widgets, view hierarchy, stroke drawing, and input handling. Every modern GUI toolkit provides these pieces in some form. Microsoft Windows, for example, has widgets (e.g., buttons, menus, text boxes), a view hierarchy (consisting of windows and child windows), a stroke drawing package (GDI), pixel representations (called bitmaps), and input handling (messages sent to a window procedure).

13.7 Toolkit Examples

	MS Win	Swing	HTML
components	windows	JComponents	elements
strokes	GDI	Graphics	-- (none)
pixels	bitmaps	Image	inlined images
input	messages -> window proc	listeners	Javascript event handlers
widgets	button, menu, textbox, …	JButton, JMenu, …	form controls & links

Here's a comparison of three UI toolkits: low-level Microsoft Windows (which few people program in anymore); Java Swing; and HTML.

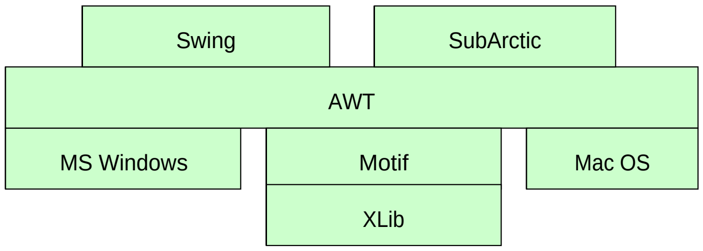
13.8 Toolkit Layering



User interface toolkits are often built on top of other toolkits, sometimes for portability or compatibility across platforms, and sometimes to add more powerful features, like a richer stroke drawing model or different widgets.

X Windows demonstrates this layering technique. The view hierarchy, stroke drawing, and input handling are provided by a low-level toolkit called XLib. But XLib does not provide widgets, so several toolkits are layered on top of XLib to add that functionality: Athena widgets and Motif, among others. More recent X-based toolkits (GTK+ and Qt) not only add widgets to XLib, but also hide XLib's view hierarchy, stroke drawing, and input handling with newer, more powerful models, although these models are implemented internally by calls to XLib.

13.9 Cross-Platform Toolkit Layering



Here's what the layering looks like for some common Java user interface toolkits.

AWT (Abstract Window Toolkit, usually pronounced like "ought") was the first Java toolkit. Although its widget set is rarely used today, AWT continues to provide drawing and input handling to more recent Java toolkits.

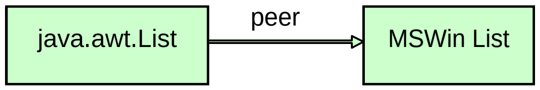
Swing is the second-generation Java toolkit, which appeared in the Java API starting in Java 1.2. Swing adds a new view hierarchy (JComponent) derived from AWT's view hierarchy (Component and Container). It also replaces AWT's widget set with new widgets that use the new view hierarchy.

subArctic was a research toolkit developed at Georgia Tech. Like Swing, subArctic relies on AWT for drawing and input handling, but provides its own widgets and views.

Not shown in the picture is SWT, IBM's Standard Widget Toolkit. (Usually pronounced "swit". Confusingly, the W in SWT means something different from the W in AWT.) Like AWT, SWT is implemented directly on top of the native toolkits. It provides different interfaces for widgets, views, drawing, and input handling.

13.9 Cross-Platform Widgets: AWT Approach

- AWT, HTML
  - Use native widgets, but only those common to all platforms
    - Tree widget available on MS Win but not X, so AWT doesn't provide it
  - Very consistent with other platform apps, because it uses the same code



Cross-platform toolkits face a special issue: should the native widgets of each platform be reused by the toolkit? One reason to do so is to preserve consistency with other applications on the same platform, so that applications written for the cross-platform toolkit look and feel like native applications. This is what we've been calling external consistency.

Another problem is that native widgets may not exist for all the widgets the cross-platform toolkit wants to provide. AWT throws up its hands at this problem, providing only the widgets that occur on every platform AWT runs on: e.g., buttons, menus, list boxes, text boxes.

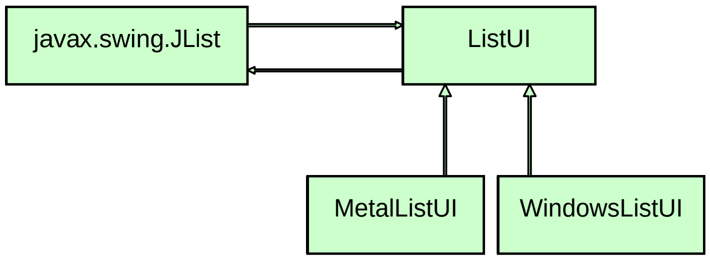
13.10 Cross-Platform Widgets: Swing approach

- Swing, Amulet
  - Reimplement all widgets
  - Not constrained by least common denominator
  - Consistent behavior for application across platforms

One reason NOT to reuse the native widgets is so that the application looks and behaves consistently with itself across platforms – a variant of internal consistency, if you consider all the instantiations of an application on various platforms as being part of the same system. Cross-platform consistency makes it easier to deliver a well-designed, usable application on all platforms – easier to write documentation and training materials, for example. Java Swing provides this by reimplementing the widget set using its default ("Metal") look and feel. This essentially creates a Java "platform", independent of and distinct from the native platform.

13.11 Pluggable Look-and-Feel

- Swing also reimplements platform look- and-feel



But Swing also supports external consistency – you can change the appearance and behavior of its widgets to make them resemble the native platform widgets. This is possible because each Swing widget delegates its painting and input handling to a look and feel object. Different sets of look-and- feel objects copy the appearance and behavior of different platforms, like Windows, Macintosh, and Motif. Unfortunately there's a lot involved in copying look and feel, and some of these platforms are moving targets – so Swing's Windows look and feel has lagged behind the changes being made in the Windows environment, making Swing applications stand out.

13.12 Cross-Platform Widgets: SWT Approach

- SWT
  - Use native widgets where available
  - Reimplement missing native widgets

Recall that AWT (and HTML) only offer widgets that are available on all platforms. SWT takes the opposite tack; instead of limiting itself to the **intersection** of the native widget sets, SWT strives to provide the **union** of the native widget sets. SWT uses a native widget if it's available, but reimplements it if it's missing, attempting to match the "style" of the platform in the new implementation as much as possible.

13.13 A Novel Toolkit: Piccolo

- Toolkit for zooming user interfaces
  - Components
    - View hierarchy is actually a **graph**
    - Components can translate, rotate, scale
    - Parents transform but **don't clip** their children by default
  - Strokes
    - Swing Graphics
  - Pixel
    - Swing Images
  - Input
    - BasicInput for mouse and keyboard events
    - Dragging controller
    - Built-in controllers for panning and zooming cameras
  - Widgets
    - PText (multiline text), PImage (images), PPath (shapes)
    - PCamera (viewport), PLayer (composite)

Finally, let's look at Piccolo, a novel UI toolkit developed at University of Maryland. Piccolo is specially designed for building **zoomable** interfaces, which use smooth animated panning and zooming around a large space. We can look at Piccolo in terms of the various aspects we've discussed in this lecture.

Layering: First, Piccolo is a layered toolkit: it runs on top of Java Swing. It also runs on top of .NET, making it a cross-platform toolkit. Piccolo ignores the platform widgets entirely, making no attempt to reimplement or reuse them. (An earlier version of Piccolo, called Jazz, could reuse Swing widgets.)

Components: Piccolo has a view hierarchy consisting of PNode objects. The hierarchy is not merely a tree, but in fact a graph: you can install camera objects in the hierarchy which act as viewports to other parts of the hierarchy, so a component may be seen in more than one place on the screen. Another distinction between Piccolo and other toolkits is that every component has an arbitrary transform relative to its parent's coordinate system – not just translation (which all toolkits provide), but also rotation and scaling. Furthermore, in Piccolo, parents do not clip their children by default. If you want this behavior, you have to request it by inserting a special clipping object (a component) into the hierarchy. As a result, components in Piccolo have two bounding boxes – the bounding box of the node itself (getBounds()), and the bounding box of the node's entire subtree (getFullBounds()).

Strokes: Piccolo uses the Swing Graphics package, augmented with a little information such as the camera and transformations in use.

Pixels: Piccolo uses Swing images for direct pixel representations.

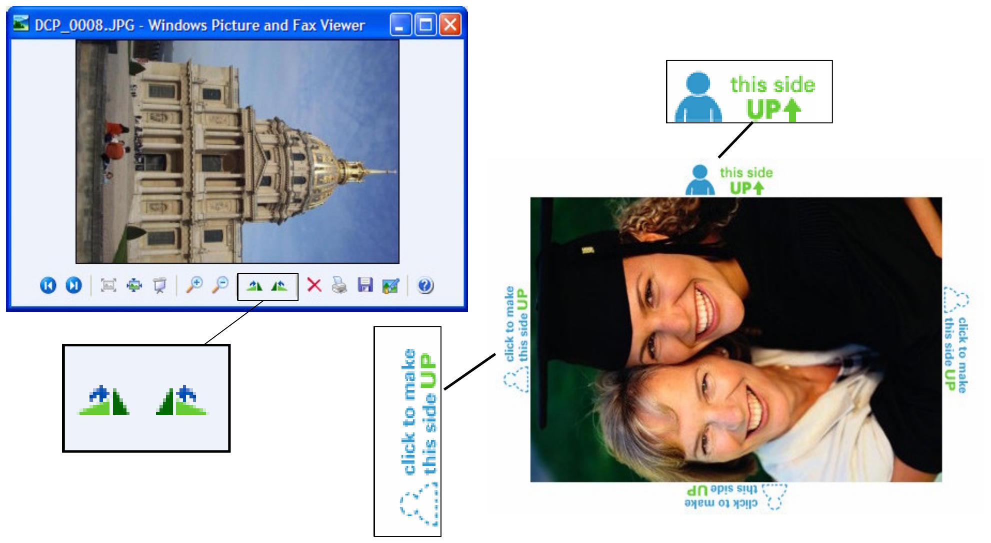
Input: Piccolo has the usual mouse and keyboard input (encapsulated in a single event-handling interface called BasicInput), plus generic controllers for common operations like dragging, panning, and zooming. By default, panning and zooming is attached to any camera you create: dragging with the left mouse button moves the camera view around, and dragging with the right mouse button zooms in and out.

Widgets: the widget set for Piccolo is fairly small by comparison with toolkits like Swing and .NET, probably because Piccolo is a research project with limited resources. It's worth noting, however, that Piccolo provides reusable components for shapes (e.g. lines, rectangles, ellipses, etc), which in other toolkits would require revering to the stroke model.

Piccolo home page: http://www.cs.umd.edu/hcil/piccolo/
Overview: http://www.cs.umd.edu/hcil/piccolo/learn/patterns.shtml
API documentation: http://www.cs.umd.edu/hcil/jazz/learn/piccolo/doc-1.1/api/

Lecture 14: Heuristic Evaluation

14.1 UI Hall of Fame or Shame?


DCP_0008.JPG - Windows Picture and Fax Viewer
this side UP
click to make this side UP

For today's UI Hall of Fame and Shame, we'll focus on the Rotate commands in photo browsers and drawing editors. These commands rotate an image by 90 degree increments, either clockwise or counterclockwise.

In the Windows XP Image Viewer, the rotation commands are represented by toolbar buttons. Unfortunately, the icons on these buttons don't work well. They're very similar to each other, and the arrow doesn't stand out (poor **contrast**). The icon tells a little story, showing before and after representations of a simplified abstract object. That's not such a bad thing in general, but it obscures the important differences between the two icons and forces you to study them carefully to figure out what they mean. Worse, the **mapping** is backwards: the Rotate Right button (with the right-pointing arrow) actually appears on the left.

The Snapfish web site (for storing and printing digital photo albums) has a neat solution to this problem. It does away with the notion of rotating entirely; instead, you just click on the side of the photo that you want to be on top. A little head-and-shoulders icon provides an **affordance** for clicking, while reminding about the heads-up orientation. This interface is neat because the controls are **directly mapped** to their effect (the side of the image that becomes the top). There's no need to mention right or left, clockwise or counterclockwise, or 90 or 180 degrees. The rotation is done by direct manipulation of the image itself. The labels are unfortunate – particularly the unreadable upside-down label! -- but new idioms often need extra help at first.

14.2 Nielsen's Heuristics

- **Meet expectations**
  1. Match the real world
  2. Consistency & standards
  3. Help & documentation
- **User is boss**
  4. User control & freedom
  5. Visibility of system status
  6. Flexibility & efficiency
- **Errors**
  7. Error prevention
  8. Recognition, not recall

9. Error reporting, diagnosis, and recovery
- **Keep it simple**

  10. Aesthetic & minimalist design

Recall these 10 heuristics we discussed in an earlier lecture.

14.3 Heuristic Evaluation

- Performed by an expert
- Steps
  - Inspect UI thoroughly
  - Compare UI against heuristics
  - List usability problems
    - Explain & justify each problem with heuristics

One application of these 10 heuristics is a usability inspection process called heuristic evaluation. Heuristic evaluation was originally invented by Jakob Nielsen, and you can learn more about it on his web site. Nielsen has done a number of studies to evaluate the effectiveness of heuristic evaluation. Those studies have shown that heuristic evaluation's cost-benefit ratio is quite favorable; the cost per problem of finding usability problems in an interface is generally cheaper than alternative methods.

Heuristic evaluation is an inspection method. It is performed by a usability expert – someone who knows and understands the heuristics we've just discussed, and has used and thought about lots of interfaces.

The basic steps are simple: the evaluator inspects the user interface thoroughly, judges the interface on the basis of the heuristics we've just discussed, and makes a list of the usability problems found – the ways in which individual elements of the interface deviate from the usability heuristics.

The Hall of Fame and Hall of Shame discussions we have at the beginning of each class are informal heuristic evaluations. In particular, if you look back at previous lecture notes, you'll see that most of the usability problems are justified by appealing to a heuristic.

14.4 How To Do Heuristic Evaluation

- Justify every problem with a heuristic
  - "Too many choices on the home page (Aesthetic & Minim la ist Design)"
  - Can't just say "I don't like the colors"
- List every problem
  - Even if an interface element has multiple problems
- Go through the interface at least twice
  - Once to get the feel of the system
  - Again to focus on particular interface elements
- Don't limit yourself to the 10 heuristics
  - We've seen others: affordances, visibility, Fitts's Law, perceptual fusion, color principles
  - But the 10 heuristics are easier to compare against

Let's look at heuristic evaluation from the evaluator's perspective. That's the role you'll be adopting in the next homework, when you'll serve as heuristic evaluators for each others' computer prototypes.

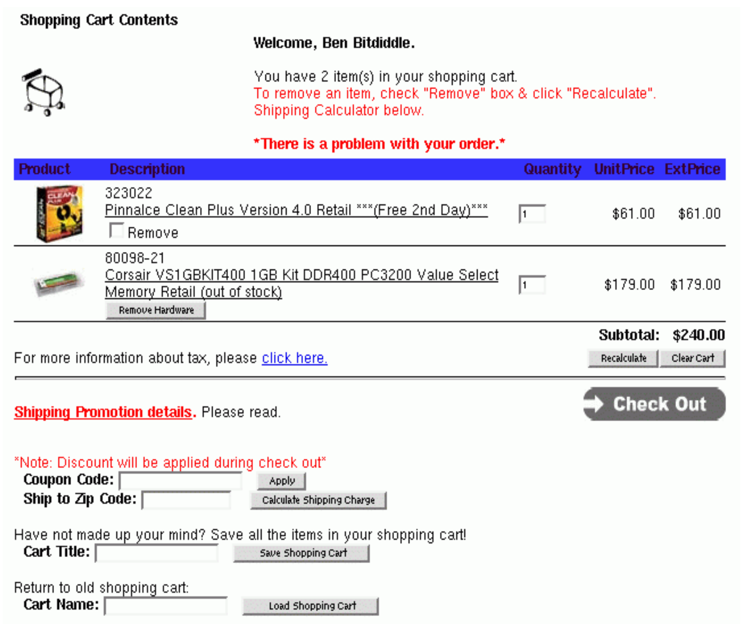
Here are some tips for doing a good heuristic evaluation. First, your evaluation should be grounded in known usability guidelines. You should justify each problem you list by appealing to a heuristic, and explaining how the heuristic is violated. This practice helps remove most of the (inevitable) subjectivity involved in inspections: You can't just say "that's an ugly yellow color." (If it's really yucky, you should pass that subjective opinion back to the design team, but you'll be forced to identify it as subjective if you can't find a heuristic to justify it.)

List every problem you find. If a button has several problems with it – inconsistent placement, bad color combination, confusing label – then each of those problems should be listed separately. Some of the problems may be more severe than others, and some may be easier to fix than others. It's best to get all the problems on the table in order to make these tradeoffs.

Inspect the interface at least twice. The first time you'll get an overview and a feel for the system. The second time, you should focus carefully on individual elements of the interface, one at a time.

Finally, although you have to justify every problem with a guideline, you don't have to limit yourself to the Nielsen 10. We've seen a number of specific usability principles that can serve equally well: affordances, visibility, Fitts's Law, perceptual fusion, color guidelines, graphic design rules are a few. The Nielsen 10 are helpful in that they're a short list that covers a wide spectrum of usability problems. For each element of the interface, you can quickly look down the Nielsen list to guide your thinking.

14.5



Shopping Cart Contents

Welcome, Ben Bitdiddle.

You have 2 item(s) in your shopping cart.
To remove an item, check "Remove" box & click "Recalculate".
Shipping Calculator below.

There is a problem with your order.

Product	Description	Quantity	UnitPrice	ExtPrice
	323022 Pinnalce Clean Plus Version 4.0 Retail ***(Free 2nd Day)*** Remove	1	$61.00	$61.00
	80098-21 Corsair VS1GBKIT400 1GB Kit DDR400 PC3200 Value Select Memory Retail (out of stock) Remove Hardware	1	$179.00	$179.00

Subtotal: $240.00

For more information about tax, please click here.

Shipping Promotion details. Please read.

Note: Discount will be applied during check out
Coupon Code:

Ship to Zip Code:

Have not made up your mind? Save all the items in your shopping cart!
Cart Title:

Return to old shopping cart:
Cart Name:

Let's try it on an example. Here's a partial heuristic evaluation of the screen shown above. Can you find any other usability issues?

1. Shopping cart icon is not balanced with its background whitespace (Aesthetic & minimalist design)
2. **Good:** user is greeted by name (Visibility of system status)
3. Red is used both for help messages and for error messages (Consistency, Match real world)
4. "There is a problem with your order", but no explanation or suggestions for resolution (Error reporting)
5. ExtPrice and UnitPrice are strange labels (Match real world)
6. Remove Hardware button inconsistent with Remove checkbox (Consistency)
7. "Click here" is unnecessary (Aesthetic & minimalist design)
8. No "Continue shopping" button (User control & freedom)
9. Recalculate is very close to Clear Cart (Error prevention)
10. "Check Out" button doesn't look like other buttons (Consistency, both internal & external)
11. Uses "Cart Title" and "Cart Name" for the same concept (Consistency)
12. Must recall and type in cart title to load (Recognition not recall, Error prevention, Flexibility & efficiency)

14.6 Heuristic Evaluation Is Not User Testing

- Evaluator is not the user either
  - Maybe closer to being a typical user than you are, though
- Analogy: code inspection vs. testing
- HE finds problems that UT often misses
  - Inconsistent fonts
  - Fitts's Law problems
- But UT is the gold standard for usability

Heuristic evaluation is only one way to evaluate a user interface. User testing -- watching users interact with the interface – is another. User testing is really the gold standard for usability evaluation. An interface has usability problems only if real users have real problems with it, and the only sure way to know is to watch and see.

A key reason why heuristic evaluation is different is that an evaluator is not a typical user either! They may be closer to a typical user, however, in the sense that they don't know the system model to the same degree that its designers do. And a good heuristic evaluator tries to think like a typical user. But an evaluator knows too much about user interfaces, and too much about usability, to respond like a typical user.

So heuristic evaluation is not the same as user testing. A useful analogy from software engineering is the difference between code inspection and testing.

Heuristic evaluation may find problems that user testing would miss (unless the user testing was extremely expensive and comprehensive). For example, heuristic evaluators can easily detect problems like inconsistent font styles, e.g. a sans-serif font in one part of the interface, and a serif font in another. Adapting to the inconsistency slows down users slightly, but only extensive user testing would reveal it. Similarly, a heuristic evaluation might notice that buttons along the edge of the screen are not taking proper advantage of the Fitts's Law benefits of the screen boundaries, but this problem might be hard to detect in user testing.

14.7 Hints for Better Heuristic Evaluation

- Use multiple evaluators
  - Different evaluators find different problems
  - The more the better, but diminishing returns
  - Nielsen recommends 3-5 evaluators
- Alternate heuristic evaluation with user testing
  - Each method finds different problems
  - Heuristic evaluation is cheaper
- It's OK for observer to help evaluator
  - As long as the problem has already been noted
  - This wouldn't be OK in a user test

Now let's look at heuristic evaluation from the designer's perspective. Assuming I've decided to use this technique to evaluate my interface, how do I get the most mileage out of it?

First, use more than one evaluator. Studies of heuristic evaluation have shown that no single evaluator can find all the usability problems, and some of the hardest usability problems are found by evaluators who find few problems overall (Nielsen, "Finding usability problems through heuristic evaluation", CHI '92). The more evaluators the better, but with diminishing returns: each additional evaluator finds fewer new problems. The sweet spot for cost-benefit, recommended by Nielsen based on his studies, is 3-5 evaluators.

One way to get the most out of heuristic evaluation is to alternate it with user testing in subsequent trips around the iterative design cycle. Each method finds different problems in an interface, and heuristic evaluation is almost always cheaper than user testing. Heuristic evaluation is particularly useful in the tight inner loops of the iterative design cycle, when prototypes are raw and low-fidelity, and cheap, fast iteration is a must.

In heuristic evaluation, it's OK to help the evaluator when they get stuck in a confusing interface. As long as the usability problems that led to the confusion have already been noted, an observer can help the evaluator get unstuck and proceed with evaluating the rest of the interface, saving valuable time. In user testing, this kind of personal help is totally inappropriate, because you want to see how a user would really behave if confronted with the interface in the real world, without the designer of the system present to guide them. In a user test, when the user gets stuck and can't figure out how to complete a task, you usually have to abandon the task and move on to another one.

14.8 Formal Evaluation Process

1. Training
   - Meeting for design team & evaluators
   - Introduce application
   - Explain user population, domain, scenarios
2. Evaluation
   - Evaluators work separately
   - Generate written report, or oral comments recorded by an observer
   - Focus on generating problems, not on ranking their severity yet
   - 1-2 hours per evaluator
3. Severity Rating
   - Evaluators prioritize all problems found (not just their own)
   - Take the mean of the evaluators' ratings
4. Debriefing
   - Evaluators & design team discuss results, brainstorm solutions

Here's a formal process for performing heuristic evaluation.

The training meeting brings together the design team with all the evaluators, and brings the evaluators up to speed on what they need to know about the application, its domain, its target users, and scenarios of use. The evaluators then go off and evaluate the interface separately. They may work alone, writing down their own observations, or they may be observed by a member of the design team, who records their observations (and helps them through difficult parts of the interface, as we discussed earlier). In this stage, the evaluators focus just on generating problems, not on how important they are or how to solve them.

Next, all the problems found by all the evaluators are compiled into a single list, and the evaluators rate the severity of each problem. We'll see one possible severity scale in the next slide. Evaluators can assign severity ratings either independently or in a meeting together. Since studies have found that severity ratings from independent evaluators tend to have a large variance, it's best to collect severity ratings from several evaluators and take the mean to get a better estimate.

Finally, the design team and the evaluators meet again to discuss the results. This meeting offers a forum for brainstorming possible solutions, focusing on the most severe (highest priority) usability problems.

When you do heuristic evaluations in this class, I suggest you follow this ordering as well: first focus on generating as many usability problems as you can, then rank their severity, and then think about solutions.

14.9 Severity Ratings

- Contributing factors
  - Frequency: how common?
  - Impact: how hard to overcome?
  - Persistence: how often to overcome?
- Severity scale
  1. Cosmetic: need not be fixed
  2. Minor: needs fixing but low priority
  3. Major: needs fixing and high priority
  4. Catastrophic: imperative to fix

Here's one scale you can use to judge the severity of usability problems found by heuristic evaluation. It helps to think about the factors that contribute to the severity of a problem: its frequency of occurrence (common or rare); its impact on users (easy or hard to overcome), and its persistence (does it need to be overcome once or repeatedly). A problem that scores highly on several contributing factors should be rated more severe than another problem that isn't so common, hard to overcome, or persistent.

14.10 Evaluating Prototypes

- Heuristic evaluation works on:
  - Sketches
  - Paper prototypes
  - Unstable prototypes
- "Missing-element" problems are harder to find on sketches
  - Because you're not actually using the interface, you aren't blocked by feature's absence
  - Look harder for them

A final advantage of heuristic evaluation that's worth noting: heuristic evaluation can be applied to interfaces in varying states of readiness, including unstable prototypes, paper prototypes, and even just sketches. When you're evaluating an incomplete interface, however, you should be aware of one pitfall. When you're just inspecting a sketch, you're less likely to notice missing elements, like buttons or features essential to proceeding in a task. If you were actually interacting with an active prototype, essential missing pieces rear up as obstacles that prevent you from proceeding. With sketches, nothing prevents you from going on: you just turn the page. So you have to look harder for missing elements when you're heuristically evaluating static sketches or screenshots.

14.11 Writing Good Heuristic Evaluations

- Heuristic evaluations must communicate well to developers and managers
- Include positive comments as well as criticisms
  - "Good: Toolbar icons are simple, with good contrast and few colors (minimalist design)"
- Be tactful
  - Not: "the menu organization is a complete mess"
  - Better: "menus are not organized by function"
- Be specific
  - Not: "text is unreadable"
  - Better: "text is too small, and has poor contrast (black text on dark green background)"

Here are some tips on writing good heuristic evaluations. First, remember your audience: you're trying to communicate to developers. Don't expect them to be experts on usability, and keep in mind that they have some ego investment in the user interface. Don't be unnecessarily harsh.

Although the primary purpose of heuristic evaluation is to identify problems, positive comments can be valuable too. If some part of the design is good for usability reasons, you want to make sure that aspect doesn't disappear in future iterations.

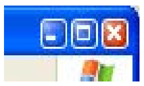
14.12 Suggested Report Format

- What to include:
  - Problem
  - Heuristic
  - Description
  - Severity
  - Recommendation (if any)
  - Screenshot (if helpful)

12. Severe: **User may close window without saving data** (error prevention)

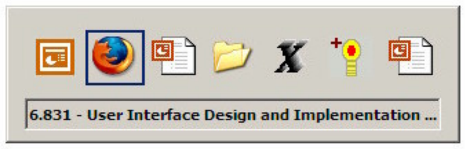
    If the user has made changes without saving, and then closes the window using the Close button, rather than File >> Exit, no confirmation dialog appears.

    Recommendation: show a confirmation dialog or save automatically



Lecture 15: User Testing

15.1 UI Hall of Fame or Shame?



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.

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. This spring- loading also provides good **dialog closure**.

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.)

There is one common operation that Alt-Tab supports wonderfully: toggling back and forth between two windows.

15.2 Today's Topics

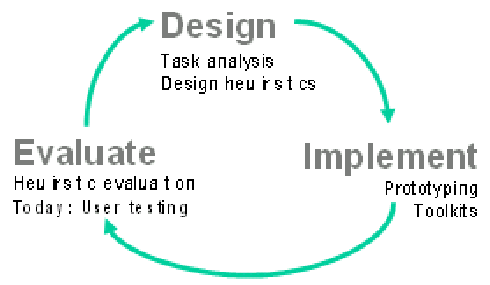
- User testing
- Ethics
- Formative evaluation

In this lecture and the next one, we'll talk about user testing: putting an interface in front of real users. There are several kinds of user testing, but all of them by definition involve human beings, who are thinking, breathing individuals with rights and feelings. When we enlist the assistance of real people in interface testing, we take on some special responsibilities. So first we'll talk about the ethics of user testing, which apply regardless of what kind of user test you're doing.

The rest of the lecture will focus on one particular kind of user test: formative evaluation, which is a user test performed during iterative design with the goal of finding usability problems to fix on the next design iteration.

Next lecture, we'll look at another kind of user test, a controlled experiment.

15.3 Review



Here's a quick review of the iterative design process, and the parts of it we've seen so far.

The only evaluation technique we've discussed so far has been heuristic evaluation. Today we're looking at user testing, which is more expensive and time-consuming than heuristic evaluation, but produces better results.

15.4 Kinds of User Tests

- Formative evaluation
  - Find problems for next iteration of design
  - Evaluates prototype or implementation, in lab, on chosen tasks
  - Qualitative observations (usability problems)
- Field study
  - Find problems in context
  - Evaluates working implementation, in real context, on real tasks
  - Mostly qualitative observations
- Controlled experiment
  - Tests a hypothesis (e.g., interface X is faster than interface Y)
  - Evaluates working implementa it on, in controlled lab environment, on chosen tasks
  - Mostly quantitative observations (time, error rate, satisfaction)

Here are three common kinds of user tests.

You've already done a **formative evaluation**, on Prototype Testing Day, when you had some of your classmates test your paper protoypes. The purpose of formative evaluation is finding usability problems in order to fix them in the next design iteration. Formative evaluation doesn't need a full working implementation, but can be done on a variety of prototypes. This kind of user test is usually done in an environment that's under your control, like an office or a usability lab. You also choose the tasks given to users, which aregenerally realistic (drawn from task analysis, which is based on observation) but nevertheless fake. The results of formative evaluation are largely **qualitative observations**, usually a list of usability problems.

Note that Prototype Testing Day was not the best way to do formative evaluation: first, because your classmates are probably not representative of your target user population; and second, because we had artificial time constraints that raised the pressure on users and experimenters, prevented using substantial tasks, and didn't allow for much debriefing or discussion after the test. Better user tests would be use appropriate users and be more relaxed, which we'll see later in the lecture.

A key problem with formative evaluation is that you have to control too much. Running a test in a lab environment on tasks of your invention may not tell you enough about how well your interface will work in a real context on real tasks. A **field study** can answer these questions, by actually deploying a working implementation to real users, and then going out to the users' real environment and observing how they use it. We won't say much about field studies in this class.

A third kind of user test is a **controlled experiment**, whose goal is to test a quantifiable hypothesis about one or more interfaces. Controlled experiments happen under carefully controlled conditions using carefully-designed tasks – often more carefully chosen than formative evaluation tasks. Hypotheses can only be tested by quantitative measurements of usability, like time elapsed, number of errors, or subjective satisfaction. We'll talk about how to design controlled experiments in the next lecture.

15.5 Ethics of User Testing

- Users are **human beings**
  - Human subjects have been seriously abused in the past
    - Nazi concentration camps
    - Tuskegee syphilis study
    - MIT Fernald School study: feeding radioactive isotopes to mentally retarded children
    - Yale electric shock study
  - Research involving user testing is now subject to close scrutiny
    - MIT Committee on Use of Humans as Experimental Subjects (COUHES) must approve research-related user studies

Let's start by talking about some issues that are relevant to all kinds of user testing: ethics. Human subjects have been horribly abused in the name of science over the past century. Here are some of the most egregious cases:

In Nazi concentration camps (1940-1945), doctors used prisoners of war, political prisoners, and Jews as human guinea pigs for horrific experiments. Some experiments tested the limits of human endurance in extreme cold, low pressures, or exposure. Other experiments intentionally infected people with massive doses of pathogens, such as typhus; others tested new chemical weapons or new medical procedures. Thousands of people were killed by these "experiments."

In the Tuskegee Institute syphilis study (1932-1972), the US government studied the effects of untreated syphilis in black men in the rural South. In exchange for their participation in the study, the men were given free health examinations. But they weren't told that they had syphilis, or that the disease was potentially fatal. Nor were they given treatment for the disease, even as proven, effective treatments like penicillin became available. Out of 339 men studied, 28 died directly of syphilis, 100 of related complications. 40 wives were infected, and 19 children were born with congenital syphilis.

In the 1940s and 1950s, MIT researchers cooperated with the Fernald School for mentally disabled children in Waverly, Massachusetts to gave radioactive isotopes to some of the children in their milk and cereal, to study how the isotopes were taken up by the body. Permission letters were obtained from their parents, but neither parents nor children were warned that radioactive materials were being used.

In the 1950s, a famous study done at Yale told subjects to give painful electric shocks to another person. The shocks weren't real, and the person they were shocking was just an actor. But subjects weren't told that fact in advance, and many subjects were genuinely traumatized by the experience: sweating, trembling, stuttering.

These abuses have led to several reforms. The Nazi-era experiments led to the Nuremberg Code, an international agreement on the rights of human subjects. The Tuskegee study drove the US government to take steps to ensure that all federally-funded institutions follow ethical practices in their use of human subjects. In particular, every experiment involving human subjects must be reviewed and approved by an ethics committee, usually called an institutional review board. MIT's review board is called COUHES.

15.6 Pressures on a User

- Performance anxiety
- Feels like an intelligence test
- Comparing self with other subjects
- Feeling stupid in front of observers
- Competing with other subjects

Experiments involving medical treatments or electric shocks are one thing. But what's so dangerous about a computer interface?

Hopefully, nothing – most user testing has minimal physical or psychological risk to the user. But user testing does put psychological pressure on the user. The user sits in the spotlight, asked to perform unfamiliar tasks on an unfamiliar (and possibly bad!) interface, in front of an audience of strangers (at least one experimenter, possibly a roomful of observers, and possibly a video camera). It's natural to feel some performance anxiety, or stage fright. "Am I doing it right? Do these people think I'm dumb for not getting it?" A user may regard the test as a psychology test, or more to the point, an IQ test. They may be worried about getting a bad score. Their self-esteem may suffer, particularly if they blame problems they have on themselves, rather than on the user interface.

A programmer with an ironclad ego may scoff at such concerns, but these pressures are real. Jared Spool, an influential usability consultant, tells a story about the time he saw a user cry during a user test. It came about from an accumulation of mistakes on the part of the experimenters:

1. the originally-scheduled user didn't show up, so they just pulled an employee out of the hallway to do the test;
2. it happened to be her first day on the job;
3. they didn't tell her what the session was about;
4. she not only knew nothing about the interface to be tested (which is fine and good), but also nothing about the domain – she wasn't in the target user population at all;
5. the observers in the room hadn't been told how to behave (i.e., shut up);
6. one of those observers was her boss;
7. the tasks hadn't been pilot tested, and the first one was actually impossible.

When she started struggling with the first task, everybody in the room realized how stupid the task was, and burst out laughing – at their own stupidity, not hers. But she thought they were laughing at her, and she burst into tears. (story from Carolyn Snyder, Paper Prototyping)

15.7 Treat the User With Respect

- Time
  - Don't waste it
- Comfort
  - Make the user comfortable
- Informed consent
  - Inform the user as fully as possible
- Privacy
  - Preserve the user's privacy
- Control
  - The user can stop at any time

The basic rule for user testing ethics is **respect** for the user as a intelligent person with free will and feelings. We can show respect for the user in 5 ways:

1. Respecting their **time** by not wasting it. Prepare as much as you can in advance, and don't make the user jump through hoops that you aren't actually testing. Don't make them install the software or load the test files, for example, unless your test is supposed to measure the usability of the installation process or file-loading process.
2. Do everything you can to make the user **comfortable**, in order to offset the psychological pressures of a user test.
3. Give the user as much **information** about the test as they need or want to know, as long as the information doesn't bias the test. Don't hide things from them unnecessarily.
4. Preserve the user's **privacy** to the maximum degree. Don't report their performance on the user test in a way that allows the user to be personally identified.
5. The user is always in **control**, not in the sense that they're running the user test and deciding what to do next, but in the sense that the final decision of whether or not to participate remains theirs, throughout the experiment. Just because they've signed a consent form, or sat down in the room with you, doesn't mean that they've committed to the entire test. A user has the right to give up the

test and leave at any time, no matter how inconvenient it may be for you.

15.8 Before a Test

- Time
  - Pilot-test all materials and tasks
- Comfort
  - "We're testing the system; we're not testing you."
  - "Any difficulties you encounter are the system's fault. We need your help to find these problems."
- Privacy
  - "Your test results will be completely confidential."
- Information
  - Brief about purpose of study
  - Inform about audiotaping, videotaping, other observers
  - Answer any questions beforehand (unless biasing)
- Control
  - "You can stop at any time."

Let's look at what you should do before, during, and after a user test to ensure that you're treating users with respect.

Long before your first user shows up, you should pilot-test your entire test: all questionnaires, briefings, tutorials, and tasks. Pilot testing means you get a few people (usually your colleagues) to act as users in a full-dress rehearsal of the user test. Pilot testing is essential for simplifying and working the bugs out of your test materials and procedures. It gives you a chance to eliminate wasted time, streamline parts of the test, fix confusing briefings or training materials, and discover impossible or pointless tasks. It also gives you a chance to practice your role as an experimenter. Pilot testing is essential for every user test.

When a user shows up, you should brief them first, introducing the purpose of the application and the purpose of the test. To make the user comfortable, you should also say the following things (in some form):

- "Keep in mind that we're testing the computer system. We're not testing you." (comfort)
- "The system is likely to have problems in it that make it hard to use. We need your help to find those problems." (comfort)
- "Your test results will be completely confidential." (privacy)
- "You can stop the test and leave at any time." (control)

You should also inform the user if the test will be audiotaped, videotaped, or watched by hidden observers. Any observers actually present in the room should be introduced to the user.

At the end of the briefing, you should ask "Do you have any questions I can answer before we begin?" Try to answer any questions the user has. Sometimes a user will ask a question that may bias the experiment: for example, "what does that button do?" You should explain why you can't answer that question, and promise to answer it after the test is over.

15.9 During the Test

- Time
  - Eliminate unnecessary tasks
- Comfort
  - Calm, relaxed atmosphere
  - Take breaks in long session
  - Never act disappointed
  - Give tasks one at a time
  - First task should be easy, for an early success experience
- Privacy

  - User's boss shouldn't be watching
- Information
  - Answer questions (again, where they won't bias)
- Control
  - User can give up a task and go on to the next
  - User can quit entirely

During the test, arrange the testing environment to make the user comfortable. Keep the atmosphere calm, relaxed, and free of distractions. (We failed on all three counts at Prototype Testing Day!) If the testing session is long, give the user bathroom, water, or coffee breaks, or just a chance to stand up and stretch.

Don't act disappointed when the user runs into difficulty, because the user will feel it as disappointment in their performance, not in the user interface.

Don't overwhelm the user with work. Give them only one task at a time. Ideally, the first task should be an easy warmup task, to give the user an early success experience. That will bolster their courage (and yours) to get them through the harder tasks that will discover more usability problems. Answer the user's questions as long as they don't bias the test.

Keep the user in control. If they get tired of a task, let them give up on it and go on to another. If they want to quit the test, pay them and let them go.

15.10 After the Test

- Comfort
  - Say what they've helped you do
- Information
  - Answer questions that you had to defer to avoid biasing the experiment
- Privacy
  - Don't publish user-identifying information
  - Don't show video or audio without user's permission

After the test is over, thank the user for their help and tell them how they've helped. It's easy to be open with information at this point, so do so.

Later, if you disseminate data from the user test, don't publish it in a way that allows users to be individually identified. Certainly, avoid using their names.

If you collected video or audio records of the user test, don't show them outside your development group without explicit written permission from the user.

15.11 Formative Evaluation

- Find some users
  - Should be representative of the target user class(es), based on user analysis
- Give each user some tasks
  - Should be representative of important tasks, based on task analysis
- Watch user do the tasks

OK, we've seen some ethical rules that apply to running any kind of user test. Now let's look in particular at how to do **formative evaluation**.

You've already done one formative evaluation test already, using your paper prototypes. So you know the basic steps already: (1) find some representative users; (2) give each user some representative tasks; and (3) watch the user do the tasks.

15.12 Roles in Formative Evaluation

- User
- Facilitator
- Observers

There are three roles in a formative evaluation test: a user, a facilitator, and some observers.

15.13 User's Role

- User should think aloud
  - What they think is happening
  - What they're trying to do
  - Why they took an action
- Problems
  - Feels weird
  - Thinking aloud may alter behavior
  - Disrupts concentration
- Another approach: pairs of users
  - Two users working together are more likely to converse naturally
  - Also called coodiscovery, constructive interaction

The user's primary role is to perform the tasks using the interface. While the user is actually doing this, however, they should also be trying to **think aloud**: verbalizing what they're thinking as they use the interface. Encourage the user to say things like "OK, now I'm looking for the place to set the font size, usually it's on the toolbar, nope, hmm, maybe the Format menu…" Thinking aloud gives you (the observer) a window into their thought processes, so you can understand what they're trying to do and what they expect.

Unfortunately, thinking aloud feels strange for most people. It can alter the user's behavior, making the user more deliberate and careful, and sometimes disrupting their concentration. Conversely, when a task gets hard and the user gets absorbed in it, they may go mute, forgetting to think aloud. One of the facilitator's roles is to prod the user into thinking aloud.

One solution to the problems of think-aloud is constructive interaction, in which two users work on the tasks together (using a single computer). Two users are more likely to converse naturally with each other, explaining how they think it works and what they're thinking about trying. Constructive interaction requires twice as many users, however, and may be adversely affected by social dynamics (e.g., a pushy user who hogs the keyboard). But it's nearly as commonly used in industry as single- user testing.

15.14 Facilitator's Role

- Does the briefing
- Pro iv des the tasks
- Coaches the user to think aloud by asking questions
  - "What are you thinking?"
  - "Why did you try that?"
- Controls the session and prevents interruptions by observers

The facilitator (also called the experimenter) is the leader of the user test. The facilitator does the briefing, gives tasks to the user, and generally serves as the voice of the development team throughout the test. (Other developers may be observing the test, but should generally keep their mouths shut.)

One of the facilitator's key jobs is to coax the user to think aloud, usually by asking general questions.

The facilitator may also move the session along. If the user is totally stuck on a task, the facilitator may progressively provide more help, e.g. "Do you see anything that might help you?", and then "What do you think that button does?" Only do this if you've already recorded the usability problem, and it seems unlikely that the user will get out of the tar pit themselves, and they need to get unstuck in order to get on to another part of the task that you want to test. Keep in mind that once you explain something, you lose the chance to find out what the user would have done by themselves.

15.15 Observer's Role

- Be quiet!
  - Don't help, don't explain, don't point out mistakes
  - Sit on your hands if it helps
- Take notes
  - affect task performance or satisfaction
  - Usually negative
    - Errors
    - Repeated attempts
    - Curses
  - May be positive
    - "Cool!"
    - "Oh, now I see."

While the user is thinking aloud, and the facilitator is coaching the think-aloud, any observers in the room should be doing the opposite: **keeping quiet**. Don't offer any help, don't attempt to explain the interface. Just sit on your hands, bite your tongue, and watch. You're trying to get a glimpse of how a typical user will interact with the interface. Since a typical user won't have the system's designer sitting next to them, you have to minimize your effect on the situation. It may be very hard for you to sit and watch someone struggle with a task, when the solution seems so obvious to you, but that's how you learn the usability problems in your interface.

Keep yourself busy by taking a lot of notes. What should you take notes about? As much as you can, but focus particularly on **critical incidents**, which are moments that strongly affect usability, either in task performance (efficiency or error rate) or in the user's satisfaction. Most critical incidents are negative. Pressing the wrong button is a critical incident. So is repeatedly trying the same feature to accomplish a task. Users may draw attention to the critical incidents with their think-aloud, with comments like "why did it do that?" or "@%!@#$!" Critical incidents can also be positive, of course. You should note down these pleasant surprises too.

Critical incidents give you a list of potential usability problems that you should focus on in the next round of iterative design.

15.16 Recording O bservations

- Pen & paper notes
  - Prepared forms can help
- Au id o recording
  - For think-aloud
- Video recording
  - Usability labs often set up with two cameras, one for user's face, one for screen
  - User may be self-conscious
  - Good for closed-circuit view by observers in another room
  - Generates too much data
  - Retrospective testing: go back through the video with the user, discussing critical incidents
- Screen capture & event logging
  - Cheap and unobtrusive
  - Camtasia, CamStudio

Here are various ways you can record observations from a user test. Paper notes are usually best, although it may be hard to keep up. Having multiple observers taking notes helps.

Audio and video recording are good for capturing the user's think-aloud, facial expressions, and body language. Video is also helpful when you want to put observers in a separate room, watching on a closed-circuit TV. Putting the observers in a separate room has some advantages: the user feels fewer eyes on them (although the video camera is another eye that can make users more self- conscious, since it's making a permanent record), the observers can't misbehave, and a big TV screen means more observers can watch. On the other hand, when the observers are in a separate room, they may not pay close attention to the test. It's happened that as soon as the user finds a usability problem, the observers start talking about how to fix that problem – and ignore the rest of the test. Having observers in the same room as the test forces them to keep quiet and pay attention. Video is also useful for retrospective testing – using the videotape to debrief the user immediately after a test. It's easy to fast forward through the tape, stop at critical incidents, and ask the user what they were thinking, to make up for gaps in think-aloud.

The problem with audio and video tape is that it generates too much data to review afterwards. A few pages of notes are much easier to scan and derive usability problems.

Screen capture software offers a cheap and easy way to record a user test, producing a digital movie (e.g. AVI or MPG). It's less obtrusive and easier to set up than a video camera, and some packages can also record an audio stream to capture the user's think-aloud.

15.17 How Many Users?

- Landauer-Nielsen model
  - Every tested user finds a fraction L of usability problems (typical L = 31%)
  - If user tests are independent, then n users will find a fraction $1-(1-L)^n$
  - So 5 users will find 85% of the problems
- Which is better:
  - Using 15 users to find 99% of problems with one design iteration
  - Using 5 users to find 85% problems with each of three design iterations
- For multiple user classes, get 3-5 users from each class

How many users do you need for formative evaluation? A simple model developed by Landauer and Nielsen ("A Mathematical Model of the Finding of Usability Problems", INTERCHI '93) postulates that every usability problem has a probability L of being found by a random user. So a single user finds a fraction L of the usability problems. If user tests are independent (a reasonable assumption if the users don't watch or talk to each other), then n users will find a fraction 1-(1-L)n of the usability problems.

Based on user tests and heuristic evaluations of 11 different interfaces, Landauer and Nielsen estimated that L is typically 31% (the actual range was 12% to 60%). With L=31%, 5 users will find about 85% of the problems.

For formative evaluation, more users is not always better. Rather than running 15 users to find almost all likely usability problems with one design iteration, it's wiser to run fewer users in each iteration, in order to squeeze in more iterations.

15.18 Flaws in Nielsen-Landauer Model

- L may be much smaller than 31%
  - Spool & Schroeder study of a CD-purchasing web site found L=8%, so 5 users only find 35% of problems
- L may vary from problem to problem
  - Different problems have different probabilities of b ie ng found, caused by:
    - Individual differences
    - Interface diversity

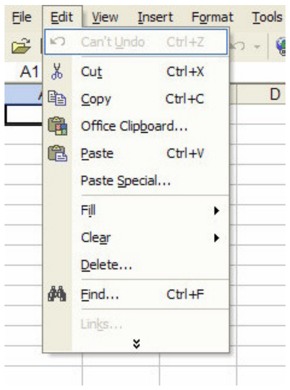
    - Task complexity
- Take-home lesson: you can't predict with confidence how many users may be needed

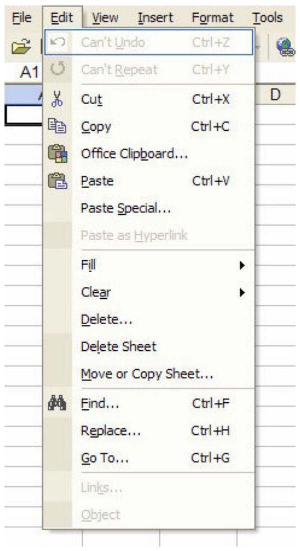
5 users is the magic number often seen in the usability literature. But L may be much smaller than 31%. A study of a website found L=8%, which means that 5 users would have found only a third of the problems (Spool & Schroeder, "Testing web sites: five users is nowhere near enough", CHI 2001). Interfaces with high diversity – different ways of doing a task – may tend to have low L values.

The probability L of finding a problem may also vary from problem to problem (and user to user). And there's no way to compute L in advance. All published values for L have been computed after the fact. There's no model for determining L for a particular interface, task, or user.

Lecture 16: Experiment Design

16.1 UI Hall of Fame or Shame?





Today's candidate for the Hall of Fame or Shame is adaptive menus, a feature of Microsoft Office. Initially, a menu shows only the most commonly used commands. Clicking on the arrow at the bottom of the menu expands it to show all available commands. Adaptive menus track how often a user invokes each command, in order to display frequently-used commands and recently-used commands.

This interface is striving for a compromise between simplicity (i.e., providing as few features as possible) and task analysis (supporting the tasks required by many users, and trying to adapt to the common tasks of each individual user). Both properties are important for usability. Unfortunately they also compete with each other. Adaptive menus are an interesting hybrid technique that's trying to satisfy both.

The downside is lack of predictability. Because the menu may change in complex and unpredictable ways – with new items appearing and underused items disappearing for no visible reason – the user can no longer rely on a lot of useful cues to find menu items. One of these cues that's lost is spatial memory – Paste may be found at different distances down the menu each time the menu appears. Another missing cue is context: Paste's neighbors may change as well.

Another downside is lack of user control. The adaptation happens automatically; the user can't manually fixate or remove items from a menu.

This particular implementation of adaptive menus has some specific usability problems. When the full menu appears, the new items are inserted into place, and there's very little contrast in the graphic design to distinguish between the old items and the new items. So the user has to search through the entire menu again.

But this particular implementation addresses other usability problems very well. When the user is hunting through all the menus, looking for a command, the full menu only has to be requested once; then all subsequent menus are fully displayed.

16.2 Today's Topics

- Experiment design

Today's lecture covers some of the issues involved in designing a controlled experiment. The issues are general to all scientific experiments, but we'll look specifically at how they apply to user interface testing.

16.3 Controled Experiment

- Start with a testable **hypothesis**
  - Interface X is faster than interface Y
- Manipulate **independent variables**
  - different interfaces, user classes, tasks
- Measure **dependent variables**
  - times, errors, satisfaction
- Use statistical tests to accept or reject the hypothesis

Here's a high-level overview of a controlled experiment. You start by stating a clear, testable hypothesis. By testable, we mean that the hypothesis must be quantifiable and measurable. For example, your hypothesis might be, "menu bars are faster than a Gimp-style right-click menu with hierarchical submenus". Here's another example that we'll use throughout this lecture: suppose you've developed two materials for the soles of children's shoes. Then your hypothesis might be, "material A wears slower than material B."

You then choose your **independent variables** – the variables you're going to manipulate in order to test the hypothesis. In our example, the independent variable is the kind of interface, menubar or right-click menu. Other independent variables may also be useful. For example, you may want to test your hypothesis on different user classes (novices and experts, or Windows users and Mac users). You may also want to test it on certain kinds of tasks. For example, in one kind of task, the menus might have an alphabetized list of names; in another, they might have functionally-grouped commands. In the shoe sole example, the independent variable would be the type of material used to make the shoe sole.

You also have to choose the **dependent variables**, the variables you'll actually measure in the experiment to test the hypothesis. Typical dependent variables in user testing are time, error rate, event count (for events other than errors – e.g., how many times the user used a particular command), and subjective satisfaction (usually measured by a questionnaire). In the shoe example, the dependent variable might be the thickness of the sole after a subject has worn it for a while. Finally, you use statistical techniques to analyze how your changes in the independent variables affected the dependent variables, and whether those effects are significant (indicating a genuine cause-and-effect) or not (merely the result of chance or noise). We'll say a little more about statistical tests in the next lecture.

16.4 Example: Fitts's Law for Menubars

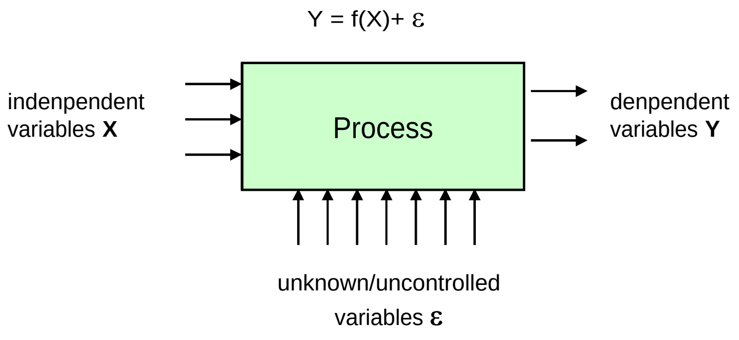
- Hypothesis: Mac menu bar is faster than Windows menu bar
- Independent: position of menu bar
- Dependent: time to reach menu bar

Here's an example of a hypothesis that we might want to test: that the Macintosh menu bar, which is anchored to the top of the screen, is faster to access than the Windows menu bar, which is separated from the top of the screen by a window title bar.

The independent variable here is the position of the menu bar: either $y = 0$ or $y = 16$ (or whatever the height of the title bar is).

The dependent variable we might measure is time: how long it takes the user to move the mouse up to the menu bar and click on a particular target menu to pull it down.

16.5 Schematic View of Experiment Design


Y = f(X)+ ε
indenpendent variables **X**
Process
denpendent variables **Y**
unknown/uncontrolled variables **ε**

Here's a block diagram to help you visualize what we're trying to do with experiment design. Think of the process you're trying to understand (e.g., menu selection) as a black box, with lots of inputs and a few outputs. A controlled experiment twiddles some of the input knobs on this box (the independent variables) and observes some of the outputs (the dependent variables) to see how they are affected.

The problem is that there are many other input knobs as well (unknown/uncontrolled variables), that may affect the process we're observing in unpredictable ways. The purpose of experiment design is to eliminate the effect of these unknown variables, or at least render harmless, so that we can draw conclusions about how the independent variables affect the dependent variables.

What are some of these unknown variables? Let's consider the shoe example. Many factors might affect the rate of wear of a shoe sole: the kind of surface walked on; the weight of the child; the way they walk (e.g., dragging their feet); their overall level of activity (sedentary or athletic); the kinds of activities they do (dancing vs. bicycling); maybe even the ambient temperature (which might soften the sole). All of these are unknown variables that might affect the dependent variable (amount of wear).

16.6 Concerns Driving Experiment Design

- Internal validity
  - Are observed results actually caused by the independent variables?
- External validity
  - Can observed results be generalized to the world outside the lab?
- Reliability
  - Will consistent results be obtained by repeating the expe ir ment?

Unknown variation is the bugaboo in experiment design, and here are the three main problems it can cause.

Internal validity refers to whether the effect we see on the experiment outputs was actually caused by the changes we made to the inputs, or caused by some unknown variable that we didn't control or measure. For example, suppose we designed the shoe experiment so that sneakers made with material A were given to boys, and sneakers made with material B were given to girls. This experiment wouldn't be internally valid, because we can't be sure whether different amounts of wear are due to the difference in materials, or to some (unknown) difference in the behavior of boys and girls. (Statisticians call this effect confounding; when a variable that we didn't control has a systematic effect on the dependent variables, it's a confounding variable.)

One way to address internal validity is to hold variables constant, as much as we can: for example, conducting all user tests in the same room, with the same lighting, the same computer, the same mouse and keyboard, the same tasks, the same training. The cost of this approach is external validity, which refers to whether the effect we see can be generalized to the world outside the lab, i.e. when those variables are not controlled. If we tried to control all the factors that might affect shoe sole wear – choosing a single surface, with one designated activity, by a single person – then it would be hard to argue that our lab experiment generalizes to how soles might wear in the varying conditions encountered in the real world.

Finally, reliability refers to whether the effect we see (the relationship between independent and dependent variables) is repeatable. If we ran the experiment again, would we see the same effect? If our experiment tested only one pair of shoes, even if we held constant every variable we could think of, unknown variations will still cause error in the experiment. A single data sample is rarely a reliable experiment.

16.7 Threats to Internal Validity

- Ordering effects
  - People learn, and people get tired
  - Don′t present tasks or interfaces in same order for all users
  - Randomize or counterbalance the ordering
- Selection effects
  - Don′t use pre-existing groups (unless group is an independent variable)
  - Randomly assign users to independent variables
- Experimenter bias
  - Experimenter may be enthusiastic about interface X but not Y
  - Give training and briefings on paper, not in person
  - Provide equivalent training for every interface
  - Double-blind experiments prevent both subject and experimenter from knowing if it′s condition X or Y
    - Essential if measurement of dependent variables requires judgement

Let's look closer at typical dangers to internal validity, and some solutions to them. You'll notice that the solutions come in two flavors: randomization (which prevents unknown variables from having systematic effects on the dependent variables) and control (which tries to hold unknown variables constant).

Ordering effects refer to the order in which different levels of the independent variables are applied. For example, does the user work with interface X first, and then interface Y, or vice versa? There are two effects from ordering: first, people learn. They may learn something from using interface X that helps them do better (or worse) with interface Y. Second, people get tired or bored. After doing many tasks with interface X, they may not perform as well on interface Y. Clearly, holding the order constant threatens internal validity, because the ordering may be responsible for the differences in performance, rather than inherent qualities of the interfaces. The solution to this threat is randomization: present the interfaces, or tasks, or other independent variables in a random order to each user.

Selection effects arise when you use pre-existing groups as a basis for assigning different levels of an independent variable. Our earlier example in which A-shoes were given to boys and B-shoes to girls was an obvious selection effect. More subtle selection effects can arise, however. Suppose you let the kids line up, and then assigned A-shoes to the first half of the line, and B-shoes to the second half. This procedure seems "random", but it isn't – the kids may line up with their friends, and groups of friends tend to have similar activities. The only safe way to eliminate selection effects is honest randomization.

A third important threat to internal validity is experimenter bias. After all, you have a hypothesis, and you're hoping it works out – you're rooting for interface X. This bias is an unknown variable that may affect the outcome, since you're personally interacting with the user whose performance you're measuring. One way to address experimenter bias is controlling the protocol of the experiment, so that it doesn't vary between the interface conditions. Provide equivalent training for both interfaces, and give it on paper, not live.

An even better technique for eliminating experimenter bias is the double-blind experiment, in which neither the subject nor the experimenter knows which condition is currently being tested. Double-blind experiments are the standard for clinical drug trials, for example; neither the patient nor the doctor knows whether the pill contains the actual experimental drug, or just a placebo. With user interfaces, double-blind tests may be impossible, since the interface condition is often obvious on its face. (Not always, though! The behavior of cascading submenus isn't obviously visible.)

Experimenter-blind tests are essential if measurement of the dependent variables requires some subjective judgement. Suppose you're developing an interface that's supposed to help people compose good memos. To measure the quality of the resulting memos, you might ask some people to evaluate the memos created with the interface, along with memos created with a regular word processor. But the memos should be presented in random order, and you should hide the interface that created each memo from the judge, to avoid bias.

16.8 Threats to External Validity

- Population
  - Draw a random sample from your real target population
- Ecological
  - Make lab conditions as realistic as possible in important respects
- Training
  - Training should mimic how real interface would be encountered and learned
- Task
  - Base your tasks on task analysis

Here are some threats to external validity that often come up in user studies. If you've done a thorough user analysis and task analysis, in which you actually went out and observed the real world, then it's easier to judge whether your experiment is externally valid.

Population asks whether the users you sampled are representative of the target user population. Do your results apply to the entire user population, or only to the subgroup you sampled? The best way to ensure population validity is to draw a random sample from your real target user population. But you can't really, because users must choose, of their own free will, whether or not to participate in a study. So there's a self-selection effect already in action. The kinds of people who participate in user studies may have unknown variables (curiosity? sense of adventure? poverty?) that threaten external validity. But that's an inevitable effect of the ethics of user testing. The best you can do is argue that on important, measurable variables – demographics, skill level, experience – your sample resembles the overall target user population.

Ecological validity asks whether conditions in the lab are like the real world. An office environment would not be an ecologically valid environment for studying an interface designed for the dashboard of a car, for example.

Training validity asks whether the interfaces tested are presented to users in a way that's realistic to how they would encounter them in the real world. A test of an ATM machine that briefed each user with a 5-minute tutorial video wouldn't be externally valid, because no ATM user in the real world would watch such a video. For a test of an avionics system in an airplane cockpit, on the other hand, even hours of tutorial may be externally valid, since pilots are highly trained.

Similarly, task validity refers to whether the tasks you chose are realistic and representative of the tasks that users actually face in the real world. If you did a good task analysis, it's not hard to argue for task validity.

16.9 Threats to Reliability

- Uncontrolled variation
  - Previous experience
    - Novices and experts: separate into different classes, or use only one class
  - User differences
    - Fastest users are 10 times faster than slowest users
  - Task design
    - Do tasks measure what you're trying to measure?
  - Measurement error
    - Time on task may include coughing, scratching, distractions
- Solutions
  - Eliminate uncontrolled variation
    - Select users for certain experience (or lack thereof)
    - Give all users the same training

- Measure dependent variables precisely

- Repetition
  - Many users, many trials
  - Standard deviation of the mean shrinks like the square root of N (i.e., quadrupling users makes the mean twice as accurate)

Once we've addressed internal validity problems by either controlling or randomizing the unknowns, reliability is what's left.

Here's a good way to visualize reliability: imagine a bullseye target. The center of the bullseye is the true effect that the independent variables have on the dependent variables. Using the independent variables, you try to aim at the center of the target, but the uncontrolled variables are spoiling your aim, creating a spread pattern. If you can reduce the amount of uncontrolled variation, you'll get a tighter shot group, and more reliable results.

One kind of uncontrolled variation is a user's previous experience. Users enter your lab with a whole lifetime of history behind them, not just interacting with computers but interacting with the real world. You can limit this variation somewhat by screening users for certain kinds of experience, but take care not to threaten external validity when you artificially restrict your user sample.

Even more variation comes from differences in ability – intelligence, visual acuity, memory, motor skills. The best users are 10 times better than the worst, an enormous variation that may swamp a tiny effect you're trying to detect.

Other kinds of uncontrolled variation arise when you can't directly measure the dependent variables. For example, the tasks you chose to measure may have their own variation, such as the time to understand the task itself, and errors due to misunderstanding the task, which aren't related to the difficulty of the interface and act only to reduce the reliability of the test. Time is itself a gross measurement which may include lots of activities unrelated to the user interface: coughing, sneezing, asking questions, responding to distractions.

One way to improve reliability eliminates uncontrolled variation by holding variables constant: e.g., selecting users for certain experience, giving them all identical training, and carefully controlling how they interact with the interface so that you can measure the dependent variables precisely. If you control too many unknowns, however, you have to think about whether you've made your experiment externally invalid, creating an artificial situation that no longer reflects the real world.

The main way to make an experiment reliable is repetition. We run many users, and have each user do many trials, so that the mean of the samples will approach the bullseye we want to hit. As you may know from statistics, the more trials you do, the closer the sample mean is likely to be to the true value. (Assuming the experiment is internally valid of course! Otherwise, the mean will just get closer and closer to the wrong value.) Unfortunately, the standard deviation of the sample mean goes down slowly, proportionally to the square root of the number of samples N. So you have to quadruple the number of users, or trials, in order to double reliability.

16.10 Blocking

- Divide samples into subsets which are more homogeneous than the whole set
  - Lots of variation between feet of different kids
  - But the feet on the same kid are far more homogeneous
  - Each child is a block
- Apply all conditions within each block
  - Put material A on one foot, material B on the other
- Measure difference within block
  - Wear(A) - Wear(B)
- Randomize within the block to eliminate internal validity threats
  - Randomly put A on left foot or right foot

Blocking is another good way to eliminate uncontrolled variation, and therefore increase reliability. The basic idea is to divide up your samples up into blocks that are more homogeneous than the whole set. In other words, even if there is lots of uncontrolled variation between blocks, the blocks should be chosen so that there is little variation within a block. Once you've blocked your data, you apply all the independent variable conditions within each block, and compare the dependent variables only within the block.

Blocking is a natural technique for the shoe sole material example. There's much uncontrolled variation between feet of different children – how they behave, where they live and walk and play – but the two feet of the same child both see very similar conditions by comparison. So we treat each child as a block, and apply one sole material to one foot, and the other sole material to the other foot. Then we measure the difference between the sole wear as our dependent variable. This technique prevents inter-child variation from swamping the effect we're trying to see.

In agriculture, blocking is done in space. A field is divided up into small plots, and all the treatments (pesticides, for example) are applied to plants in each plot. Even though different parts of the field may differ widely in soil quality, light, water, or air quality, plants in the same plot are likely to be living in very similar conditions.

Blocking helps solve reliability problems, but it doesn't address internal validity. What if we always assigned material A to the left foot, and material B to the right foot? Since most people are right- handed and left-footed, some of the difference in sole wear may be caused by footedness, and not by the sole material. So you should still randomize assignments within the block.

16.11 Between Subjects vs. Within Subjects

- "Between subjects" design
  - Users are divided into two groups:
    - One group sees only interface X
    - Other group sees only interface Y
  - Results are compared **between** different groups
    - Is mean(xi) > mean(yj)?
  - Eliminates variation due to ordering effects
    - User can't learn from one interface to do better on the other
- "Within subjects" design
  - Each user sees both interface X and Y (in random order)
  - Results are compared **within** each user
    - For user i, compute the difference xi - yi
    - Is mean(xi - yi) > 0?
  - Eliminates variation due to user differences
    - User only compared with self

The idea of blocking is what separates two commonly-used designs in user studies that compare two interfaces. Looking at these designs also gives us an opportunity to review some of the concepts

A between-subjects design is unblocked. Users are randomly divided into two groups. These groups aren't blocks! Why? First, because they aren't more homogeneous than the whole set – they were chosen randomly, after all. And second, because we're going to apply only one independent variable condition within each group. One group uses only interface X, and the other group uses only interface Y. The mean performance of the X group is then compared with the mean performance of the Y group. This design eliminates variation due to interface ordering effects. Since users only see one interface, they can't transfer what they learned from one interface to the other, and they won't be more tired on one interface than the other. But it suffers from reliability problems, because the differences between the interfaces may be swamped by the innate differences between users. As a result, you need more repetition – more users – to get reliable and significant results from a between subjects design.

A within-subjects design is blocked by user. Each user sees both interfaces, and the differential performance (performance on X – performance on Y) of all users is averaged and compared with 0. This design eliminates variation due to user differences, but may have reliability problems due to ordering effects.

Which design is better? User differences cause much more variation than ordering effects, so the between-subjects design needs more users than the within-subjects design. But the between-subjects design may be more externally valid.

16.12 Design ofthe Menubar Experiment

- Users
  - Windows users or Mac users?
  - Age, handedness?
  - How to sample them?
  - Within - or between - subjects?
- Implementation
  - Real Windows vs. real Mac
  - Artificial window manager that lets us control menu bar position
- Tasks
  - Realistic: word processing, email, web browsing
  - Artificial: repeatedly pointing at fake menu bar
- Measurement
  - When does movement start and end?
- Ordering
  - of tasks and interface conditions
- Hardware
  - mouse, trackball, touchpad, joystick?
  - PC or Mac? which particular machine?

Let's return to the menubar example, where we want to test the hypothesis that the Mac menu bar is faster to reach than the Windows menu bar. Here are some of the issues we'd have to consider in designing this study. First, what user population do we want to sample? Does experience matter? Windows users will be more experienced with one kind of menu bar, and Mac users with the other. On the other hand, the model underlying our hypothesis (Fitts's Law) is largely independent of long-term memory or practice, so we might expect that experience doesn't matter. But that's another hypothesis we should test, so maybe past experience should be an independent variable that we select when sampling.

How do we sample the user population we want? The most common technique is advertising around a college campus, but this biases against older users and less-educated users. Any sampling method has biases; you have to collect demographic information, report it, and worry about whether the bias influences your data.

Should this be a within-subjects or between-subjects study? In a within-subjects study, each user will use both the Mac menu bar and the Windows menu bar; in the between-subjects study, they'll use only one. The biggest question here is, which is larger: individual variation between users, or learning effects between interfaces used by the same user? A within-subjects design would probably be best here, since it has higher power, and we want to test expert usage, so learning effects should be minimized.

What implementation should we test? One possibility is to test the menu bars in their real context: inside the Mac interface, and inside the Windows interface, using real applications and real tasks. This gives more external validity, but the problem is now the presence of confounding variables -- the size of the menu bars might be different, the reading speed of the font, the mouse acceleration parameters, etc. We need to control for as many of these variables as we can. Another possibility is implementing our own window manager that holds these variables constant and merely changes the position of the menu bar.

What tasks should we give the user? Again, having the user use the menu bar in the context of realistic tasks might provide more external validity; but it would also be noisier. An artificial experiment that simply displays a menu bar and cues the user to hit various targets on it would provide more reliable results. But then the user's mouse would always be in the menu bar, which isn't at all realistic. We'd need to force the user to move the mouse out of the menu bar between trials, perhaps to some home location in the middle of the screen.

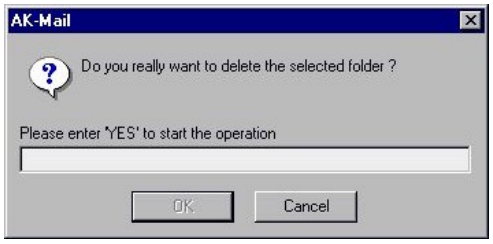
How do we measure dependent variable, time? Maybe from the time the user is given the cue ("click Edit") to the time the user successfully clicks on Edit.

What order should we present the tasks and the interface conditions, to deal with learning effects? Since we're doing a within-subjects study, we should randomize the order of interface conditions and of tasks.

Finally, the hardware we use for the study can introduce lots of confounding variables. We should use the same computer for the entire experiment.

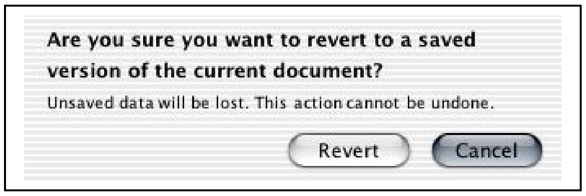
Lecture 17: Experiment Analysis

17.1 UI Hall of Fame or Shame?



Do you really want to delete the selected folder ?

Please enter 'YES' to start the operation



Are you sure you want to revert to a saved version of the current document?

Unsaved data will be lost. This action cannot be undone.

Our Hall of Fame/Shame candidates for today are **confirmation dialog boxes**.

The AK-Mail dialog at the top is almost insulting to the user's intelligence. Furthermore, it certainly isn't **efficient**, since it's forcing the user to jump through hoops in order to complete an operation. Why was this done? No doubt for **error prevention**. Deleting a folder is evidently an irreversible operation, and potentially a catastrophic one if the folder is full of important mail. So the designers didn't want the user to trigger it accidentally. Hence the confirmation dialog. But they didn't want a thoughtless press of the Enter key to accidentally confirm the confirmation dialog, either. So this dialog box requires the user to put some conscious effort into confirming it.

But this effort isn't spent thinking about the **consequences** of the operation; instead, it's spent following the directions to type YES. In fact, the dialog doesn't even explain why the user is being forced to jump through hoops.

The dialog box on the bottom does a much better job. It explains the consequences and lets the user make a decision. To avoid accidental Enter presses, it provides a safe default, Cancel.

Even better than a confirmation dialog would be a **deferred operation** – don't execute the irreversible operation immediately (although the UI should appear as if it has executed). That way, you can make the operation undoable for a little while.

17.2 Today's Topics

- Hypothesis testing
- Statistical significance
- T test
- ANOVA

17.3 An Experiment

- Hypothesis: Mac menubar is faster to access than Windows menubar
  - Design: between-subjects, randomized assignment of interface to subject

Windows	Mac
400	360
220	210
560	500
340	305

Suppose we've conducted an experiment to compare the position of the Mac menubar (flush against the top of the screen) with the Windows menubar (separated from the top by a window title bar). For the moment, let's suppose we used a **between-subjects** design. We recruited either users, and each user used only one version of the menu bar, and we'll be comparing different users' times. For simplicity, each user did only one trial, clicking on the menu bar just once while we timed their speed of access. The results of the experiment are shown above (times in milliseconds). Mac **seems** to be faster (343.75 ms on average) than Windows (380 ms). But given the noise in the measurements – some of the Mac trials are actually much slower than some of the Windows trials -- how do we know whether the Mac menubar is really faster?

This is the fundamental question underlying statistical analysis: estimating the amount of evidence in support of our hypothesis, even in the presence of noise.

17.4 Hypothesis Testing

- Our hypothesis: position of menubar matters
  - i.e., mean(Mac times) < mean(Windows times)
  - This is called the alternative hypothesis (also called H1)
- If we're wrong: position of menu bar makes no difference
  - i.e., mean(Mac) = mean(Win)
  - This is called the null hypothesis (H0)
- We can't really disprove the null hypothesis
  - Instead, we argue that the chance of seeing a difference **at least as extreme** as what we saw is very small if the null hypothesis is true

Our hypothesis is that the position of the menubar makes a difference in time. Another way of putting it is that the (noisy) process that produced the Mac access times is **different** from the process that produced the Windows access times. Let's make the hypothesis very specific: that the mean access time for the Mac menu bar is less than the mean access time for the Windows menu bar.

In the presence of randomness, however, we can't really prove our hypothesis. Instead, we can only present evidence that it's the best conclusion to draw from all possible other explanations. We have to argue against a skeptic who claims that we're wrong. In this case, the skeptic's position is that the position of the menu bar makes no difference; i.e., that the process producing Mac access times and Windows access times is the same process, and in particular that the mean Mac time is equal to the mean Windows time. This hypothesis is called the **null hypothesis**. In a sense, the null hypothesis is the "default" state of the world; our own hypothesis is called the **alternative hypothesis**.

Our goal in hypothesis testing will be to accumulate enough evidence – enough of a difference between Mac times and Windows times – so that we can **reject the null hypothesis** as very unlikely.

17.5 Statistical Significance

- Compute a statistic from our experimental data
  X = mean(Win) – mean(Mac)
- Determine the probability distribution of the statistic assuming H0 is true
  Pr(X=x | H0)
- Measure the probability of getting the same or greater difference
  Pr (X > x0 | H0)   one-sided test
  2 Pr (X > |x0| | H0)   two-sided test
- If that probability is less than 5%, then we say

    - “We reject the null hypothesis at the 5% significance level”
    - equivalently: “difference between menubars is statistically significant (p < .05)”
- Statistically significant does not mean scientifically important

Here’s the basic idea behind statistical testing. We boil all our experimental data down to a single statistic (in this case, we’d want to use the difference between the average Mac time and the average Windows time). If the null hypothesis is true, then this statistic has a certain probability distribution.

(In this case, if H0 is true and there’s no difference between Windows and Mac menu bars, then our difference in averages should be distributed around 0, with some standard deviation sigma).

So if H0 is really true, we can regard our entire experiment as a single random draw from that distribution. If the statistic we computed turned out to be a typical value for the H0 distribution – really near 0, for example – then we don’t have much evidence for arguing that H0 is false. But if the statistic is extreme – far from 0 in this case – then we can **quantify** the likelihood of getting such an extreme result. If only 5% of experiments would produce a result that’s at least as extreme, then we say that we reject the null hypothesis – and hence accept the alternative hypothesis H1, which is the one we wanted to prove – at the 5% significance level.

The probability of getting at least as extreme a result given H0 is called the **p value** of the experiment. Small p values are better, because they measure the likelihood of the null hypothesis. Conventionally, the p value must be 5% to be considered **statistically significant**, i.e. enough evidence to reject. But this convention depends on context. An experiment with very few trials (n<10) may be persuasive even if its p value is only 10%. Conversely, an experiment with thousands of trials won’t be terribly convincing unless its p value is 1% or less.

Keep in mind that **statistical significance does not imply importance**. Suppose the difference between the Mac menu bar and Windows menu bar amounts to only 1 millisecond (out of 350 milliseconds on average). A sufficiently large experiment would be able to detect this difference at the 5% significance level, but the difference is so small that it wouldn’t be useful for user interface design.

17.6 T test

- T test compares the means of two samples
- Two-sided:
    - H0: means are equal
    - H1: means are different
- One-side:
    - H0: means are equal
    - H1: mean(A) < mean(B)
- Assumptions:
    - samples A & B are independent (between- subjects, randomized)
    - normally distributed
    - equal variance

Let’s look at some of the more common statistical tests that are used in user interface experiments. The T test is what you’d use to compare two means in a between-subjects experiment, like the hypothetical Mac/Windows menubar experiment we’ve been discussing. The T statistic computes the difference between the Mac average and the Windows average, divided by an estimate of the standard deviation. If the null hypothesis is true, then this statistic follows a T distribution (which looks very similar to a normal distribution, a hump centered at 0). You can look up the value of the T statistic you computed in a table of the T distribution to find out the probability of getting a more extreme value.

There are two forms of the T test, with different alternative hypotheses. In the more conservative, **two-sided T** test, your alternative hypothesis is merely that the means are different, so an extreme t value (either positive or negative) counts as evidence against the null hypothesis. The other form is the one-sided test, in which your alternative hypothesis expects the difference to go one way or the other – e.g., if there’s any difference between Mac and Windows at all, the Mac should be faster. It’s conventional to use the **two-sided** test unless you (and the skeptic you’re arguing against) are completely certain which way the difference should go, if the difference exists at all.

Using the T test requires a few assumptions. First, your samples should be independent, so you need to use good experiment design with randomization and controls to prevent inadvertent dependence between samples. Second, the T test also assumes that the underlying probability distribution of the samples (e.g., the access times) is a normal distribution, and that even if the alternative hypothesis is true, both samples have equal variance. Fortunately the T test is not too sensitive to the normality and equal-variance assumptions: if your sample is large enough (N > 20), deviations don't affect it much.

17.7 Paired T Test

- For within-subject experiments
- Uses the mean of the differences (each user against themselves)
- H0: mean of differences is zero
- H1: mean of differences is nonzero (two-sided test)

What if we had run a **within-subjects** experiment instead? Then we would need to compare each subject with themselves, by computing the difference between each subject's Macintosh access time and the same subject's Windows access time. We would then use a t test to test the hypothesis that the mean of these differences is nonzero, against the null hypothesis that the mean of the differences is zero. This test is called a **paired t test**.

Why is a paired t test more powerful? Because by computing the difference within each user, we're canceling out the contribution that's unique to the user. That means that individual differences between users are no longer contributing to the noise (variance) of the experiment.

17.8 Analysis of Variance (ANOVA)

- Compares more than 2 means
- One-way ANOVA
  - 1 independent variable with k >= 2 levels
  - H0: all k means are equal
  - H1: the means are different (so the independent variable matters)

So far we've only looked at one independent variable (the menu bar position) with only two levels tested (Mac position and Windows position). If you want to test means when you have more than one independent variable, or more than two levels, you can use ANOVA (short for Analysis of Variance).

One-way ANOVA (also called "single factor ANOVA") addresses the case where you have more than two levels of the independent variable that you're testing. For example, suppose we wanted to test a third menu bar position at the bottom of the screen. Then we'd have three samples: top (Mac), below title (Windows), and bottom. One-way ANOVA can simultaneously compare all three means against the null hypothesis that all the means are equal.

ANOVA works by weighing the variation between the independent variable conditions (Mac vs. Windows vs. bottom) against the variation within the conditions (which is due to other factors like individual differences and random noise). If the null hypothesis is true, then the independent variable doesn't matter, so dividing up the observations according to the independent variable is merely an arbitrary labeling. Thus, assuming we randomized our experiment properly, the variation between those arbitrary groups should be due entirely to chance, and identical to the random variation within each group. So ANOVA takes the ratio between these two variations (computed as mean sums of squares), and if this ratio is significantly greater than 1, then that's sufficient evidence to argue that the null hypothesis is false and the independent variable actually does matter.

17.9 Two-Way ANOVA

- 2 independent variables with j and k levels, respectively
- Tests whether each variable has an effect independently
- Also tests for interaction between the variables

ANOVA can be extended to multiple independent variables, by looking at the variation between different levels of one independent variable (while holding the other independent variable constant). This is **two-way** (or two-factor) ANOVA.

Two-way ANOVA can be used to analyze a within-subjects experiment, where one independent variable is the variable we were testing (e.g. menubar position), while the other independent variable is the user's identity.