

Lecture 15: Usability

User Interface Hall of Shame



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

2

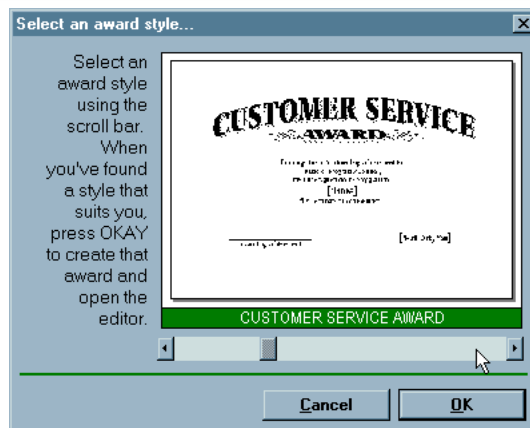
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template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

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

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

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

User Interface Hall of Shame



Source: Interface Hall of Shame

Spring 2004

6.170 Laboratory in Software Engineering

3

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.

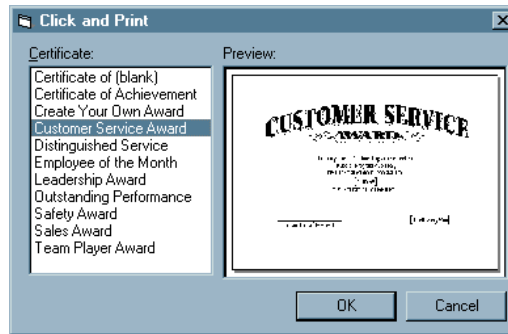
But the dialog doesn't get any better for frequent users, either. If a frequent user wants a template they've used before, how can they find it? Surely they'll remember that it's 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes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

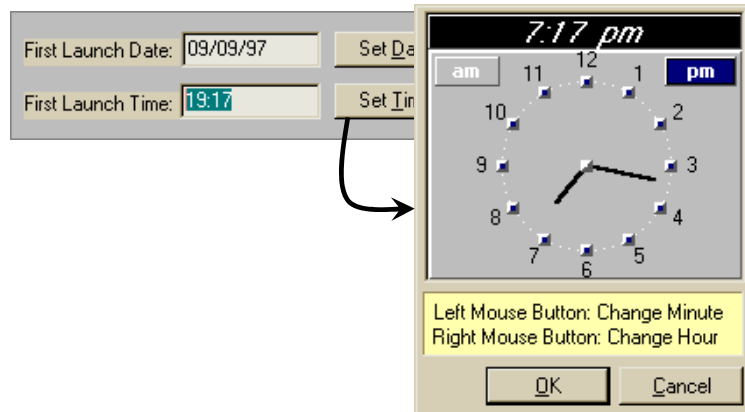
Redesigning the Interface



Source: Interface Hall of Shame

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.

Another for the Hall of Shame



Source: Interface Hall of Shame

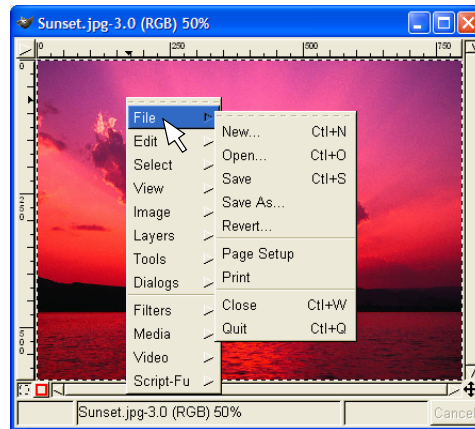
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to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

Hall of Fame or Hall of Shame?



Spring 2004

6.170 Laboratory in Software Engineering

6

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.

Gimp's designers made a choice without fully considering how it interacted with human capabilities.

Hall of Shame or Hall of Fame?



Finally, we have the much-reviled Paperclip.

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

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

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.

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).

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 statement: (1) most users don't, and (2) user customizations may be even worse! One study of command abbreviations found that users made twice as many errors with their *own* command abbreviations than with a carefully-designed set (Grudin & Barnard, "When does an abbreviation become a word?", CHI '85). So customization isn't the silver bullet.

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.

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.

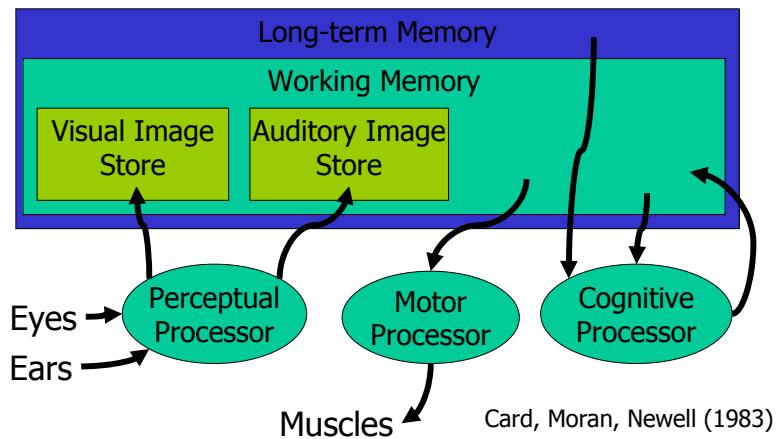
More narrowly defined, usability measures how well users can use the system's functionality. Usability has several dimensions: learnability, efficiency, memorability, error rate/severity, and subjective satisfaction.

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.

Outline

- Today: the human machine
 - Perception
 - Motor skills
 - Memory
 - Color
- Tomorrow: usability engineering

Model Human Processor



Spring 2004

6.170 Laboratory in Software Engineering

14

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 important for us to understand some of the properties of the **human** that we're designing for.

The Model Human Processor was developed by Card, Moran, and Newell as a way to summarize decades of psychology research in an **engineering model**. It's a high-level look at the cognitive abilities of a human being -- really high level, like 30,000 feet. MHP is an abstraction, of course. But it's an abstraction that actually gives us *numerical parameters* describing how we behave. (Card, Moran, Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, 1983)

Just as a computer has memory and processor, so does our model of a human. Actually, the MHP has several different kinds of memory, and several different processors.

The **perceptual processor** takes input from the eyes and ears and drops it into two temporary memories, the **visual image store** and the **auditory image store**. As a computer hardware analogy, these memories are like frame buffers, storing a single frame of perception.

The **motor processor** takes instructions from the **working memory** (which you might think of as RAM, although it's pretty small), and runs those instructions on the muscles.

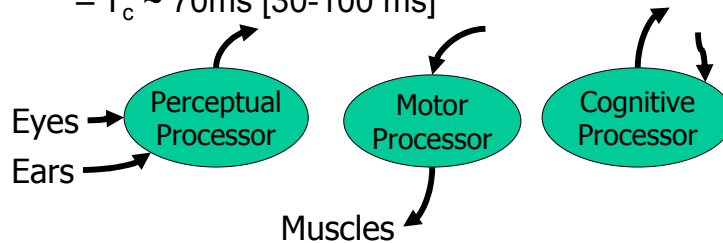
The **cognitive processor** operates on data from all the memories, including **long-term memory**, and puts its results back in the working memory.

Note that this model doesn't reflect the anatomy of your nervous system. There probably isn't an area in your brain corresponding to the perceptual processor. But it's a useful abstraction nevertheless. As an analogy, a computer's memory isn't physically an array of bytes, even though that's the abstraction we use to program it. Each byte is actually divided among several chips, 1 bit per chip, and different parts of the memory are on different DIMMs, and a couple of caches sit between the CPU and the RAM.

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

Processors in the MHP

- Processors have a cycle time
 - $T_p \sim 100\text{ms}$ [50-200 ms]
 - $T_m \sim 70\text{ms}$ [25-170 ms]
 - $T_c \sim 70\text{ms}$ [30-100 ms]



The cycle time of an MHP processor is analogous to the cycle time of a computer processor. It's the time 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, T_p , 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 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.

Perceptual Fusion

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

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 enough, but 20 frames per second is better (T_p may be as fast as 50 ms for the quickest humans and for 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.

Fitts's Law

- Open-loop control: $T_m \sim 70$ ms
- Closed-loop control: $T_p + T_c + T_m \sim 240$ ms
- Moving your hand to a target is a sequence of corrections
 - Each cycle covers remaining distance D with error ϵD
- Fitt's Law
 - Time T to move your hand to a target of size S at distance D away is:
 - $T = a + b \log(D/S + 0.5)$
 - Depends only on relative precision required D/S

Spring 2004

6.170 Laboratory in Software Engineering

17

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. We can explain Fitts's Law by appealing to the Model Human Processor.

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 $T_p + T_c + T_m \sim 240$ 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.)

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 $\epsilon^2 D$. This process repeats, with the error decreasing geometrically, until n iterations have brought your hand within the target – i.e., $\epsilon^n D \leq 1/2 S$. Solving for n , and letting the total time $T = n(T_p + T_c + T_m)$, we get $T = A \log(2D/S)$, where $A = -(T_p + T_c + T_m)/\log \epsilon$.

The upshot is that the time to move your hand to a target depends only on the *relative precision* required, D/S , not on the absolute value of either distance or size. (Note that this derivation doesn't hold well when D/S is small, because very few feedback cycles are needed to hit a very close or very large target. Experimental evidence suggests the 0.5 constant inside the log.)

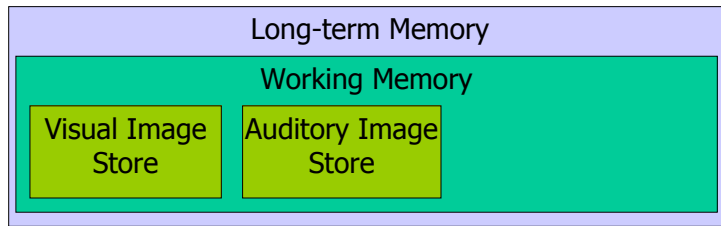
Implications of Fitts's Law

- 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 submenu
 - Windows: 500 ms timeout
 - Mac does it right: triangular zone

Fitts's Law has some easy implications:

- The edge of the screen stops the mouse pointer, so you don't need a correcting cycle to hit it. Essentially, the edge of the screen acts like a target with *infinite* size. 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.
- Now we can see why Gimp's 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. 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 now we know a reason that this solution isn't ideal: it exceeds T_p , so it destroys perceptual fusion and our sense of causality. And you still have to make that right-angle turn to get into the menu. 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. The user can point straight to the submenu without unusual corrections, and without even noticing that there might be a problem.

Memories in the MHP



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

Now we turn to the other part of the Model Human Processor: the memories. MHP memories are characterized by three properties: encoding, size, and decay time.

We won't say much about the **visual image store** or **auditory image store**, but you can think of these memories as frame buffers for visual or audio input. Their encoding has physical features (not exactly pixels, but features such as curvature, length, and edges). The VIS memory is fleeting, decaying in a few hundred milliseconds; the AIS lasts longer, a few seconds.

Memory

- Working memory
 - Encoded as **chunks**
 - MWR CAA OLI BMF BIA TMD ODP HDI SPB
 - BMW RCA AOL IBM FBI ATM DOD PHD ISP
 - Capacity: 7 ± 2 “chunks”
 - Decay: 7 [5-226] sec
- Long-term memory
 - Also encoded as chunks
 - Huge capacity
 - Little decay

Working memory is where you do your conscious thinking. In terms of the MHP, 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.

The elements of working memory and long-term memory are called **chunks**. In one sense, chunks are defined symbols; in another sense, a chunk represents the activation of past experience.

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.

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.

The capacity of working memory is roughly 7 ± 2 chunks, but this varies strongly with load.

Working memory decays in tens of seconds. Maintenance rehearsal – repeating the items to yourself – fends off this decay, much as DRAM must refresh itself. But distraction destroys working memory.

Long-term memory is probably the least understood part of the MHP. It contains the mass of our memories. Its capacity is huge, and it exhibits little decay. Long-term memories are apparently not intentionally erased; they just become inaccessible. Maintenance rehearsal (repetition) appears to be useless for moving information into long-term memory. Instead, the mechanism seems to be **elaborative rehearsal**, which seeks to make connections with existing chunks. Elaborative rehearsal lies behind the power of mnemonic techniques like associating things you need to remember with familiar places, like rooms in your childhood home.

Color Blindness

- 8% of people can't distinguish red-green

The Google logo is displayed with its standard colors: blue 'G', red 'o', yellow 'o', green 'g', and red 'le'. The letters are in a sans-serif font with a slight shadow effect.

normal vision

The Google logo is displayed as it would appear to someone with red-green color blindness. The red letters (the first 'o' and the 'le') appear as a dark purple or black, making the logo look like 'Gooogle'.

red-green deficient

- Blue-yellow color blindness also exists, but is rarer

Color deficiency (“color blindness”) affects a significant fraction of human beings. An overwhelming number of them are male. 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. For some kinds of red-green color deficiency, the red light also looks darker than the green light.

Chromatic Aberration

- Lens can't focus blue and red at same time
- So blue-on-red text looks fuzzy and hurts to read



Chromatic aberration is another important problem. The refractive index of your eye's 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.

Next Time: Usability Engineering

- Iterative design
- Low-fidelity prototypes
- Heuristics
- User testing