Lecture 1: Introduction to Usability

Prof. Rob Miller
MIT EECS

User Interface Hall of Shame

Source: Interface Hall of Shame

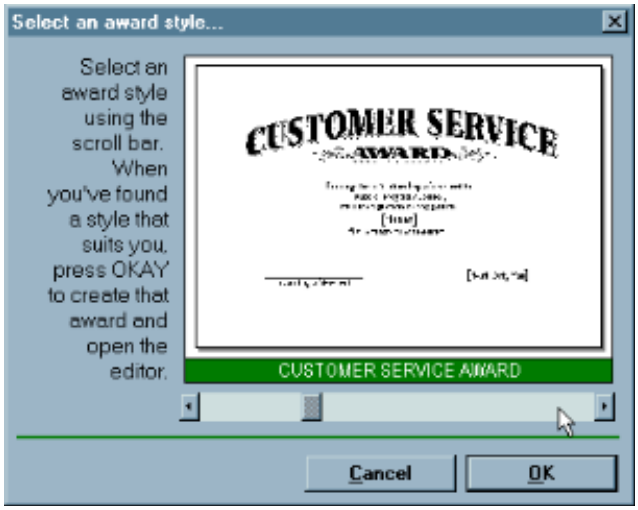
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template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

Normally, a horizontal scrollbar underneath an image (or document, or some other content) is designed for scrolling the content horizontally. A new or infrequent user looking at the window sees the scrollbar, assumes it serves that function, and ignores it. **Inconsistency** with prior experience and other applications tends to trip up new or infrequent users.

Another way to put it is that the horizontal scrollbar is an **affordance** for continuous scrolling, not for discrete selection. We see affordances out in the real world, too; a door knob says "turn me", a handle says "pull me". We've all seen those apparently-pullable door handles with a little sign that says "Push"; and many of us have had the embarrassing experience of trying to pull on the door before we notice the sign. The help text on this dialog box is filling the same role here.

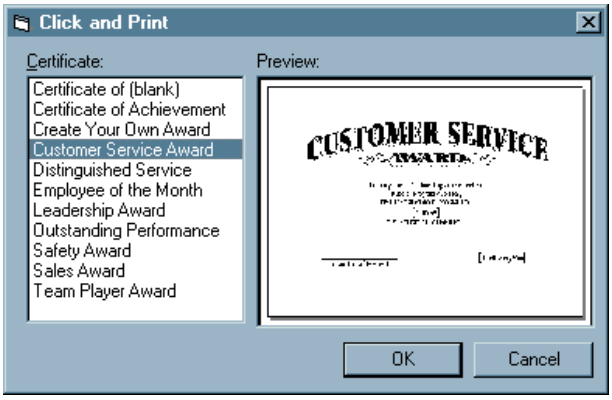
But the dialog doesn't get any better for frequent users, either. If a frequent user wants a template they've used before, how can they find it? Surely they'll remember that it's 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes what should be a random access process and transforms it into a linear process. Every user has to look through all the choices, even if they already know which one they want. The computer scientist in you should cringe at that algorithm.

Even the help text has usability problems. "Press OKAY"? Where is that? And why does the message have a ragged left margin? You don't see ragged left too often in newspapers and magazine layout, and there's a good reason.

On the plus side, the designer of this dialog box at least recognized that there was a problem – hence the help message. But the help message is indicative of a flawed approach to usability. Usability can't be left until the end of software development, like package artwork or an installer. It can't be patched here and there with extra messages or more documentation. It must be part of the process, so that usability bugs can be *fixed*, instead of merely patched.

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

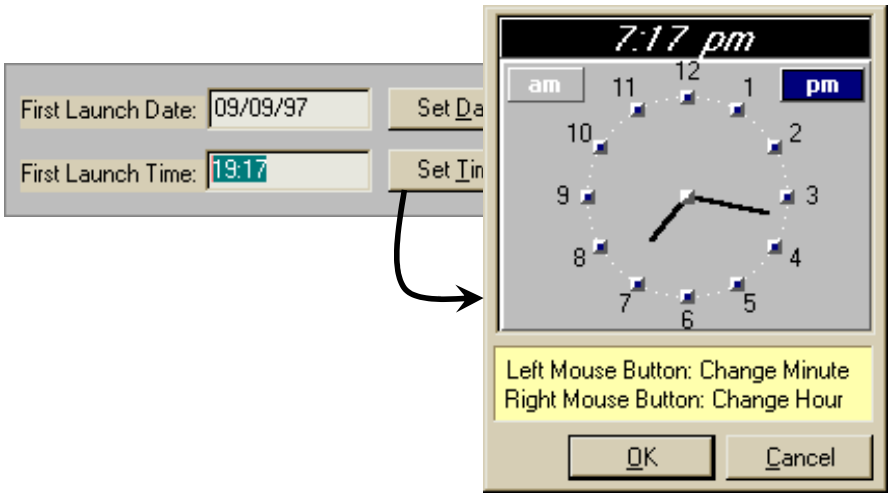
The Example, Redesigned



Source: Interface Hall of Shame

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.

More UI Hall of Shame



Source: Interface Hall of Shame

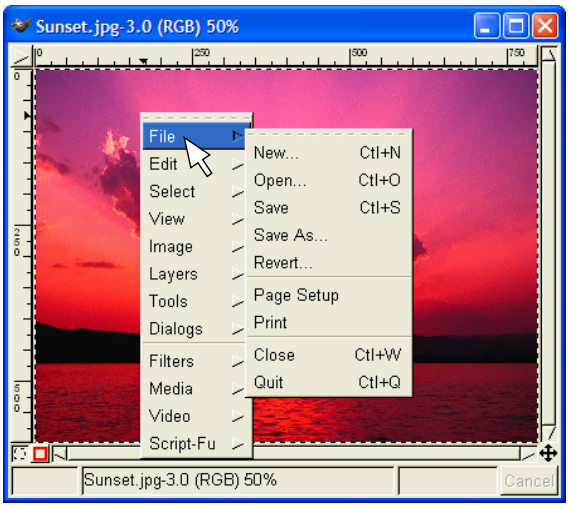
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to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

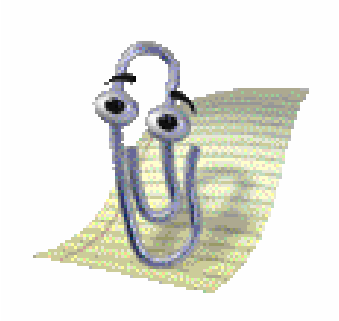
This is certainly inconsistent with other applications, and new users are likely to stumble trying to find, for example, the File menu, which never appears on a context menu in other applications. (I certainly stumbled as a new user of Gimp.) But Gimp's designers were probably thinking about expert users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

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.

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.

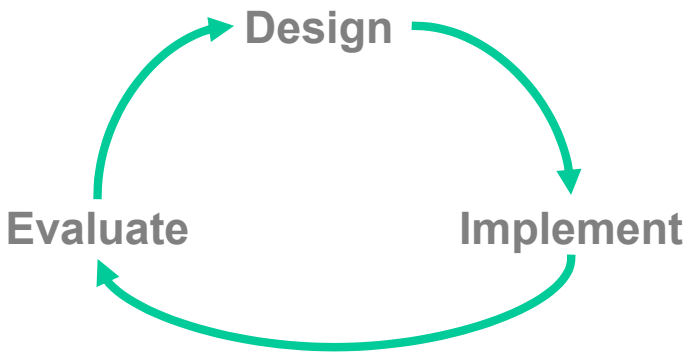
Usability Is Only One Attribute of a System

- Software designers have a lot to worry about:
  - Functionality
  - Performance
  - Cost
  - Security
  - **Usability**
  - Size
  - Reliability
  - 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.

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.

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.

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.

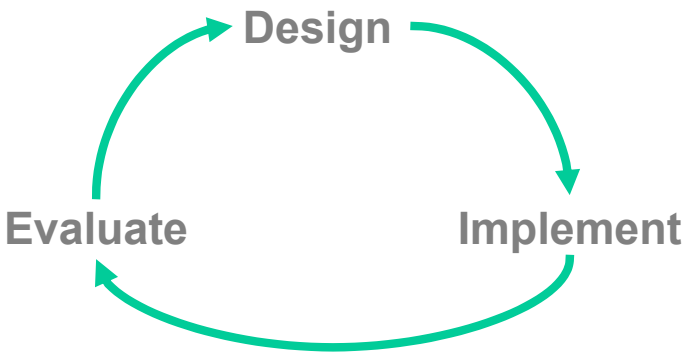
Evaluate

- 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.

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.

Goals of 6.893

- 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

Administrivia

- General information
- Lecture notes
- Problem sets
- Term project
- Readings

Readings for Next Time

- Boehm, “Spiral Model”
- Gould, “Olympic Message System”