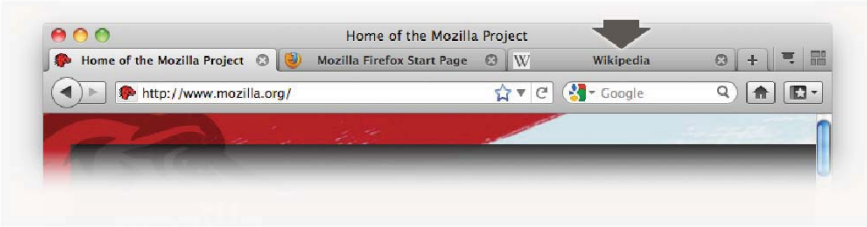
Lecture 6: User-Centered Design

GR1 (project proposal & analysis)
released today, due Sunday

Content in this lecture indicated as "All Rights Reserved" is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.

UI Hall of Fame or Shame?



© Mozilla. All rights reserved.

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

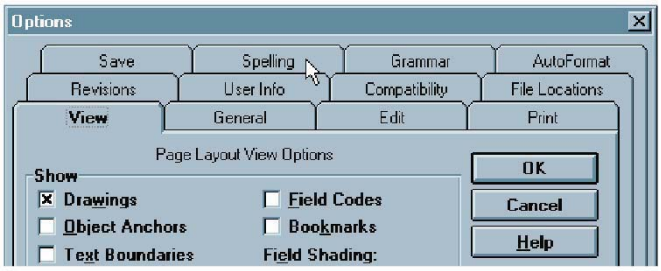
Tabbed browsing neatly solves a scaling problem in the Windows taskbar. If you accumulate several top-level windows, they cease to be separately clickable buttons in the taskbar and merge together into a single button with a popup menu. So your browser windows become **less visible** and **less efficient**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

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

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

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

Hall of Shame

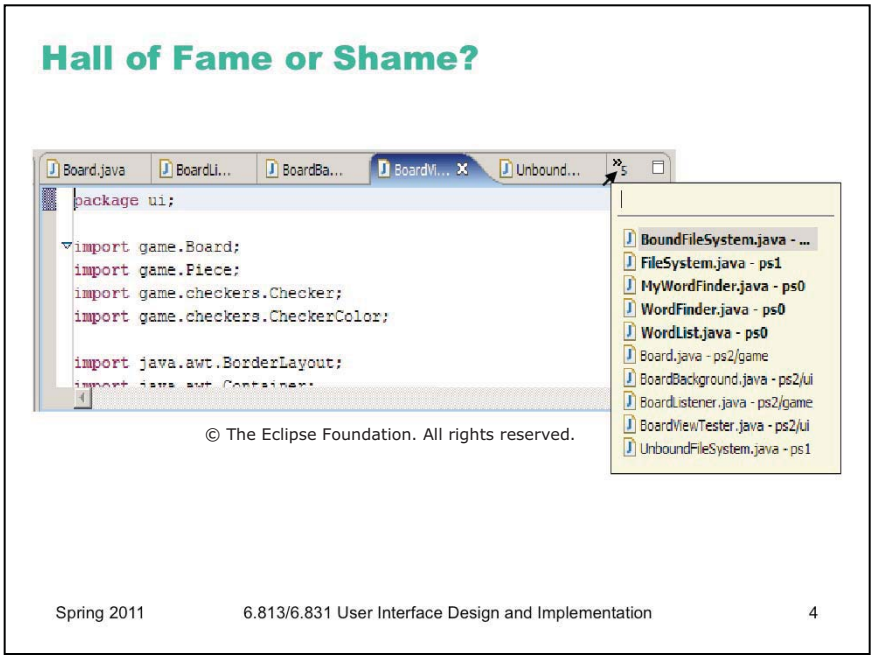


© Microsoft. All rights reserved.

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

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

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



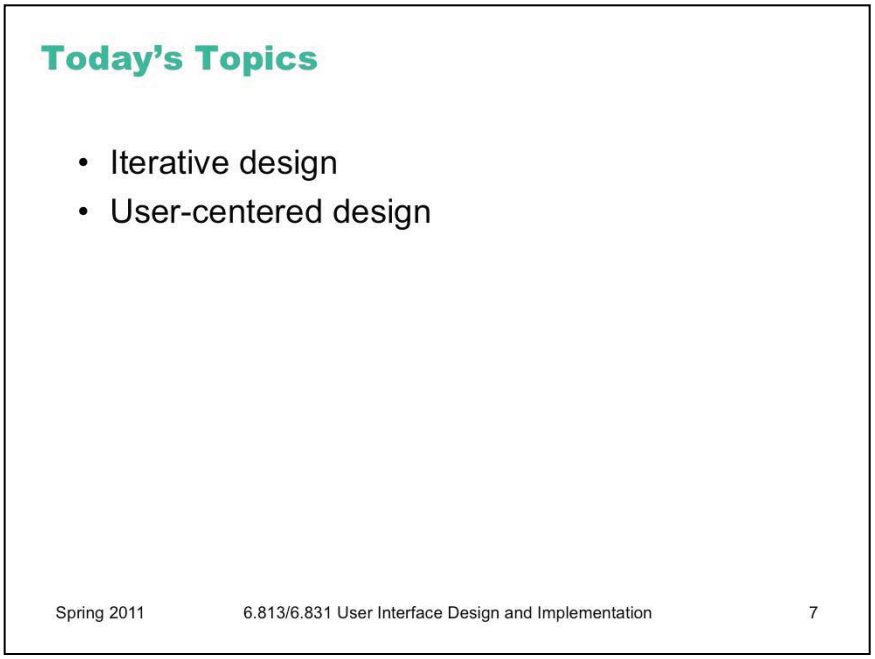
Here's how Eclipse tries to address the tab scaling problem: it shows a few tabs, and the rest are found in a pulldown menu on the right end of the tab bar.

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

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

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

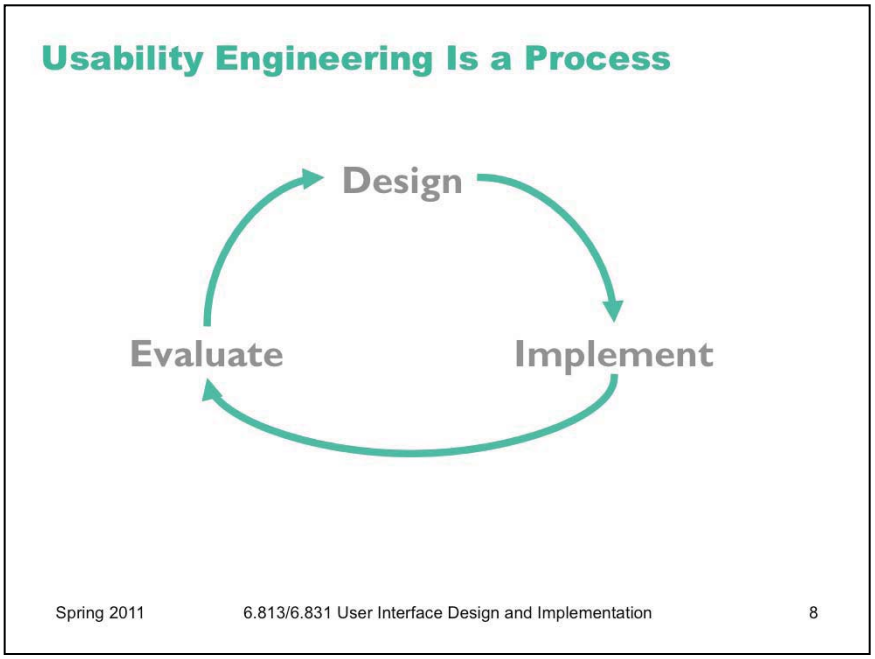
The key problem with this pulldown menu is that it completely disregards the **natural, spatial mapping** that tabs provide. The menu's order is unrelated to the order of the visible tabs; instead, the menu is alphabetical, but the tabs themselves are listed in order of recent use. If you choose a hidden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



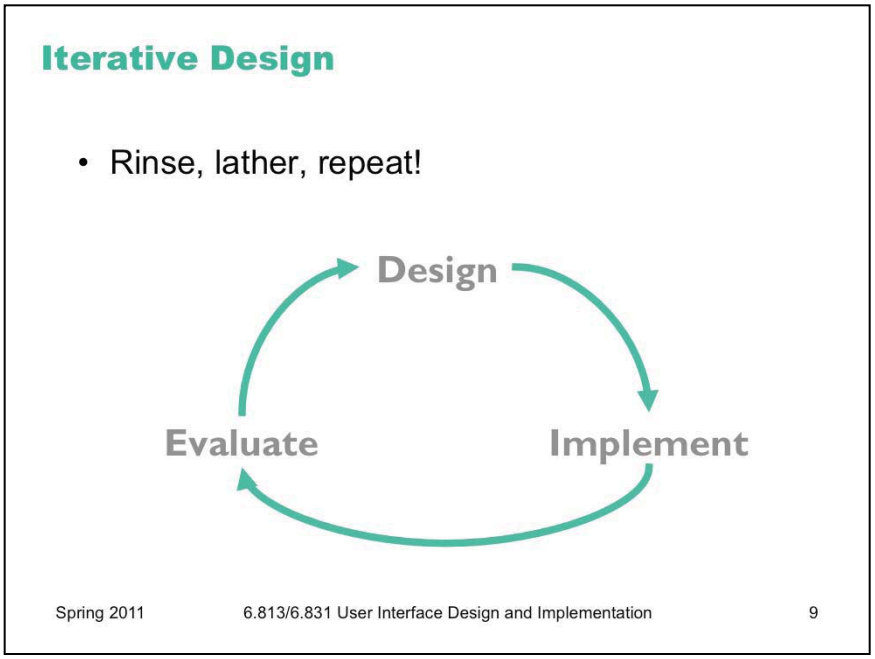
Today's lecture concerns two topics.

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.

Second, we'll look at a specific kind of iterative design called the **user-centered design process**, which is a widely-accepted way to build user interfaces with good usability properties. Your term project is structured as a user-centered design process.

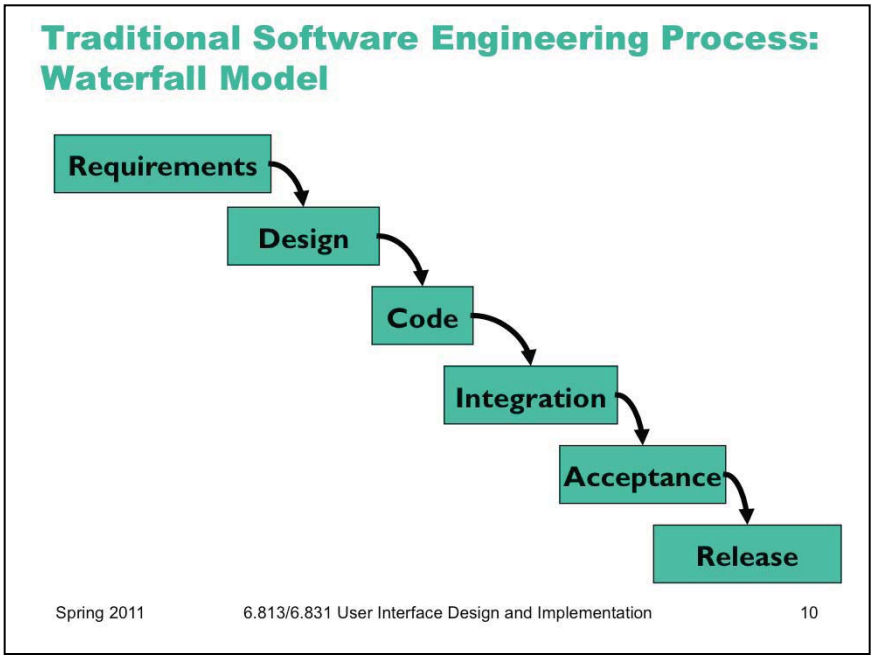


We've said that user interfaces are hard to build, and usability is a hard property to guarantee. So how do we **engineer** usability into our systems? By means of a process, with careful attention to detail. One goal of this course is to give you experience with this usability engineering process.



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

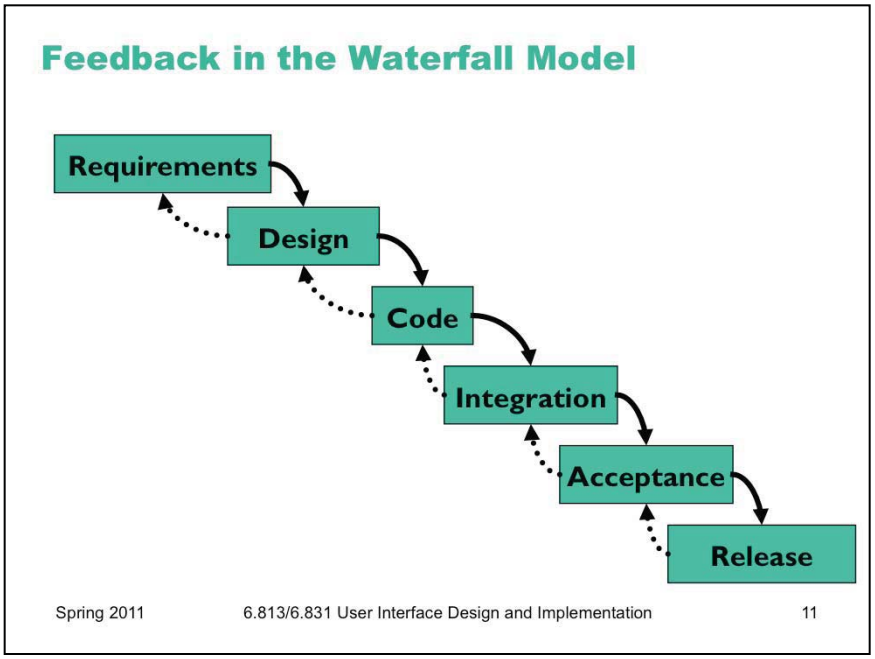
Many of the techniques we'll learn in this course are optimizations for the iterative design process: design guidelines reduce the number of iterations by helping us make better designs; cheap prototypes and discount evaluation techniques reduce the cost of each iteration. But even more important than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



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

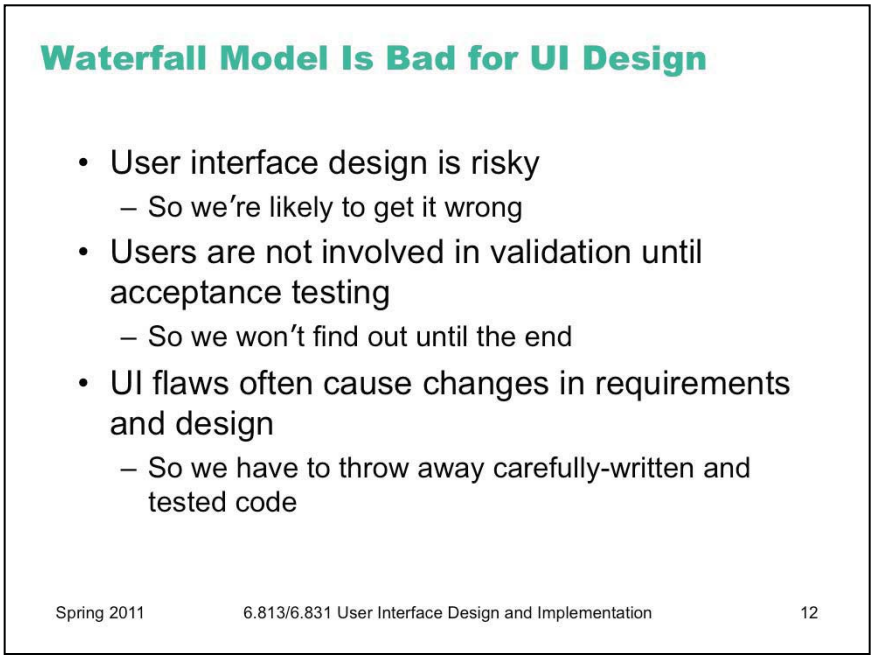
The biggest improvement of the waterfall model over previous (chaotic) approaches to software development is the discipline it puts on developers to **think first, and code second**. Requirements and designs generally precede the first line of code.

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



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

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

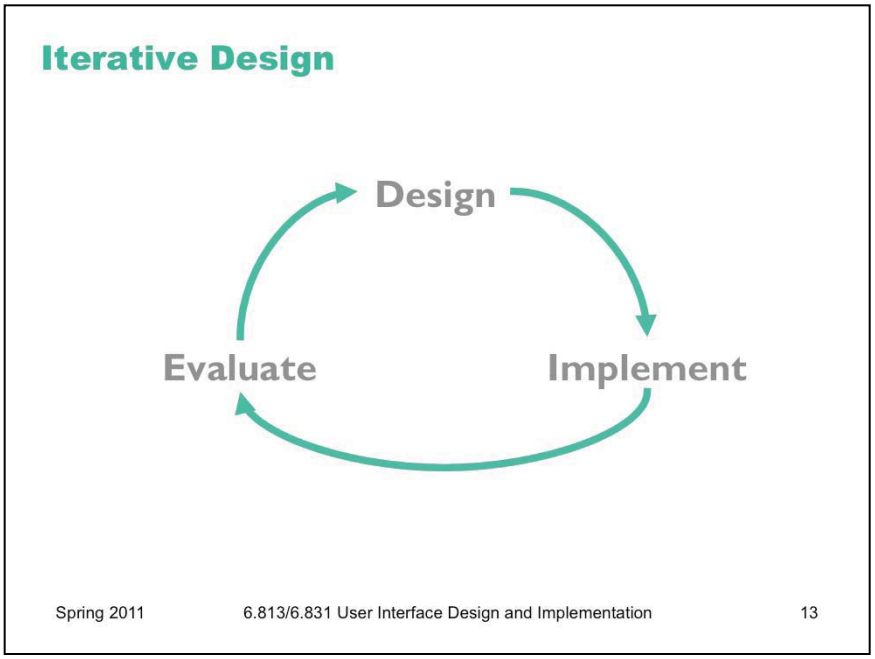


Although the waterfall model is useful for some kinds of software development, it's very poorly suited to user interface development.

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

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

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



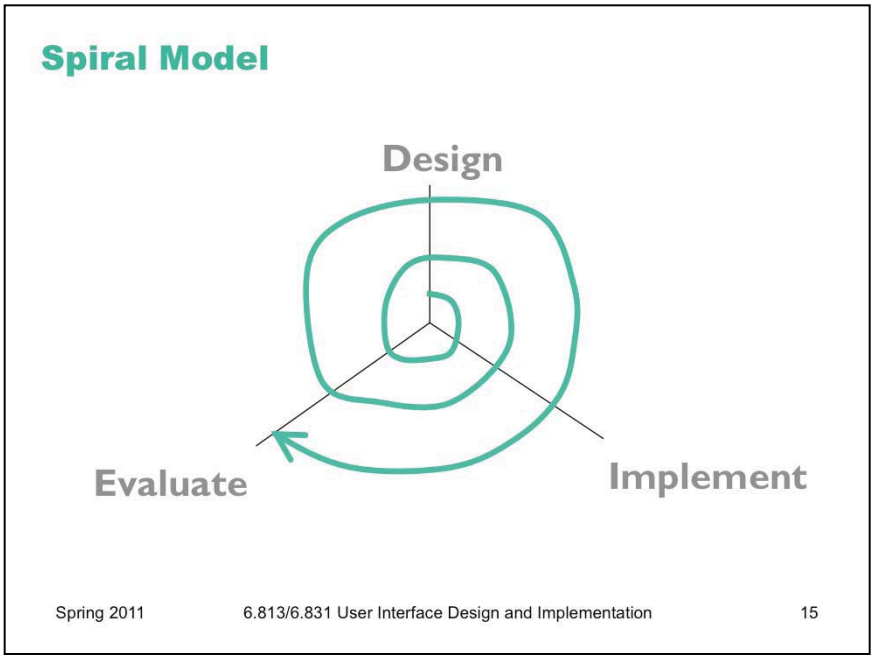
Iterative design offers a way to manage the inherent risk in user interface design. In iterative design, the software is refined by repeated trips around a design cycle: first imagining it (design), then realizing it physically (implementation), then testing it (evaluation).

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?

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

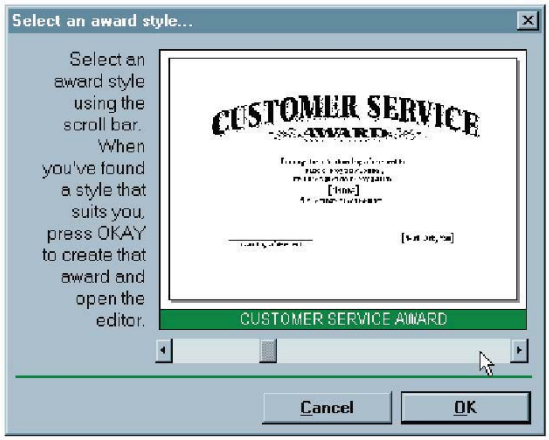
Unfortunately, many commercial UI projects have followed this model. They do a standard waterfall design, produce a bad UI, and release it. Evaluation then takes place in the marketplace, as hapless customers buy their product and complain about it. Then they iterate the design process on version 2.



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

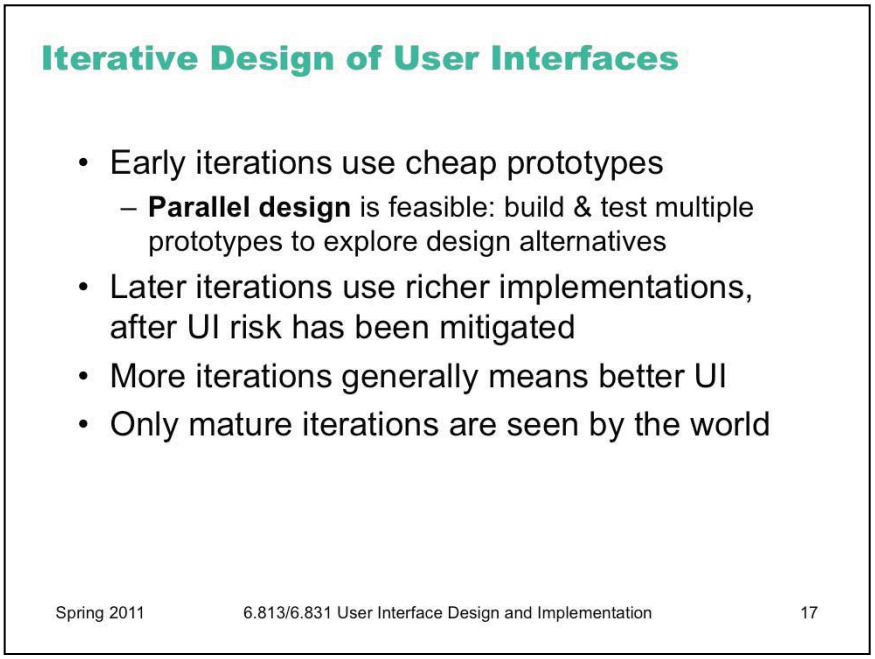
The radial dimension of the spiral model corresponds to the **cost** of the iteration step – or, equivalently, its **fidelity** or **accuracy**. For example, an early implementation might be a paper sketch or mockup. It's low-fidelity, only a pale shadow of what it would look and behave like as interactive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

Early Prototypes Can Detect Usability Problems



© Interface Hall of Shame. All rights reserved.

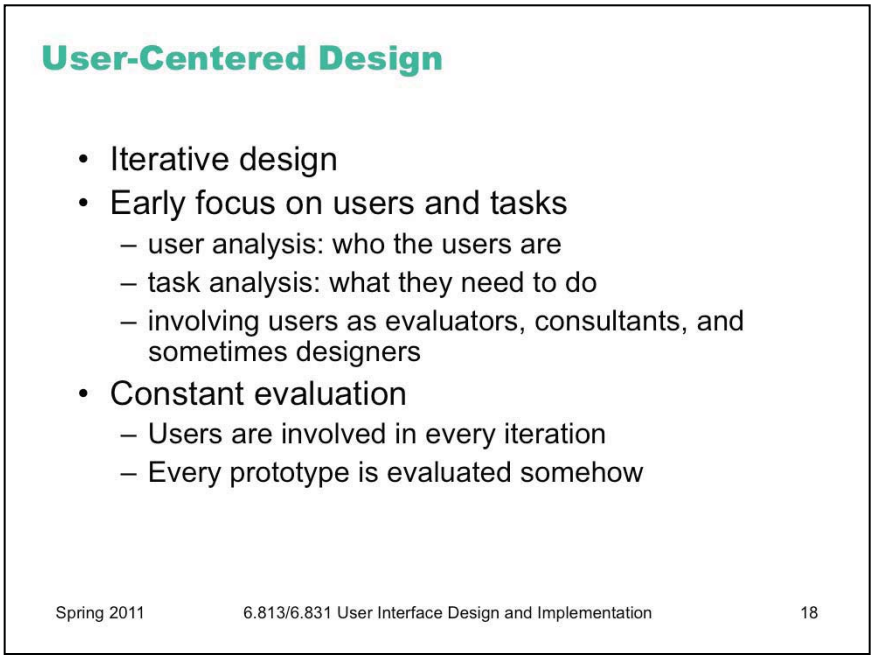
Remember this Hall of Shame candidate from the first lecture? This dialog's design problems would have been easy to catch if it were only tested as a simple paper sketch, in an early iteration of a spiral design. At that point, changing the design would have cost only another sketch, instead of a day's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



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.

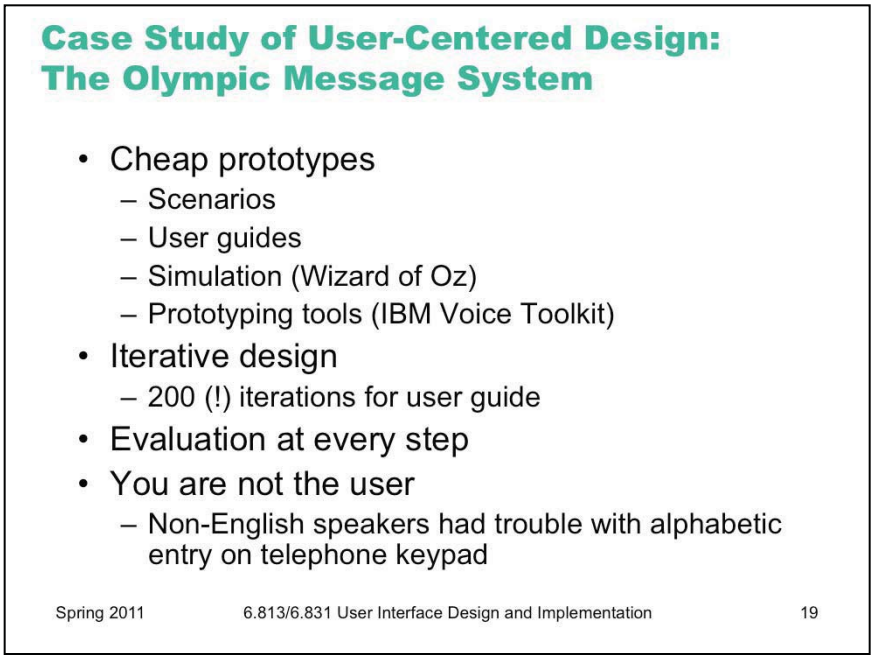


Iterative design is a crucial part of **user-centered design**, the design process for user interfaces that is widely accepted among UI practicioners. A variety of brand-name user-centered design techniques exist (e.g., GUIDE, STUDIO, OVID, LUCID). But most have three features in common:

- iterative design using rapid prototyping

- early focus on users and tasks

- evaluation throughout the iterative design process.

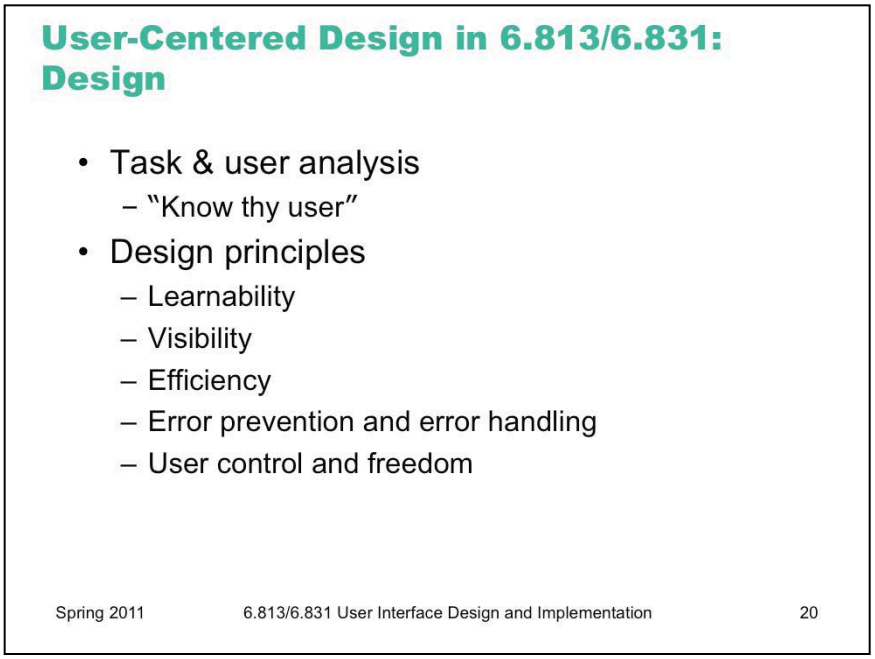


The Olympic Message System is a classic demonstration of the effectiveness of user-centered design (Gould et al, "The 1984 Olympic Message System", CACM, v30 n9, Sept 1987). The OMS designers used a variety of cheap prototypes: scenarios (stories envisioning a user interacting with the system), manuals,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

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

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

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



This course will cover many aspects of user-centered design, both in lecture content and in the group project.

The first step of user-centered design 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 the process too. We've covered these already. Design guidelines help you get started, and help avoid the most boneheaded mistakes like the ones we see in the UI Hall of Shame. 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 *visibility,* but making everything visible fights with *simplicity*. Design guidelines help us discuss design alternatives sensibly, but they don't give all the answers.

User-Centered Design in 6.813/6.831: Implementation

- Prototyping
  - Cheap, throw-away implementations
  - Low-fidelity: paper, Wizard of Oz
  - Medium-fidelity: HTML, Flash
- GUI implementation techniques
  - Output & input models
  - Desktop vs. web GUI
  - Constraints & layout

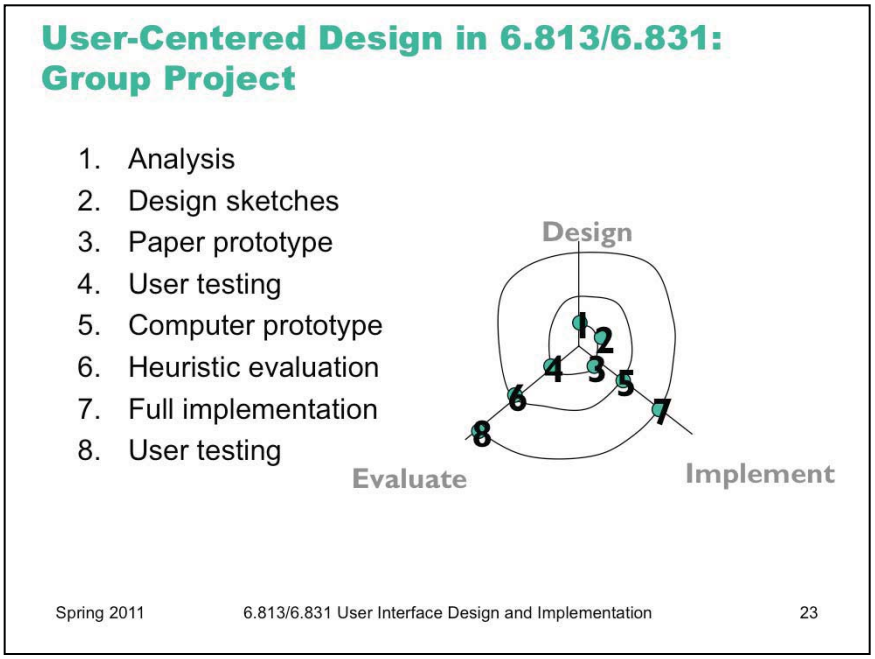
Since we can't predict usability in advance, we build **prototypes** – the cheaper, the better. We want to throw them away. We'll see that paper is a great prototyping tool, although its fidelity is low -- it can't capture everything that we might want to try. But there are richer kinds of prototyping techniques too.

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.

User-Centered Design in 6.813/6.831: Evaluation

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

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



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

1. task and user analysis (1 week): collecting the requirements for the UI, which we'll discuss in the next lecture.

2. design sketches (1 week): paper sketches of various UI designs.

3. paper prototype (1 week): an interactive prototype made of paper and other cheap physical materials.

4. user testing (1 week): during the paper prototype assignment, you'll test your prototype on your classmates.

5. computer prototype (2 weeks): an incomplete but interactive software prototype.

6. heuristic evaluation (1 week): we'll exchange implementation prototypes and evaluate them as usability experts would.

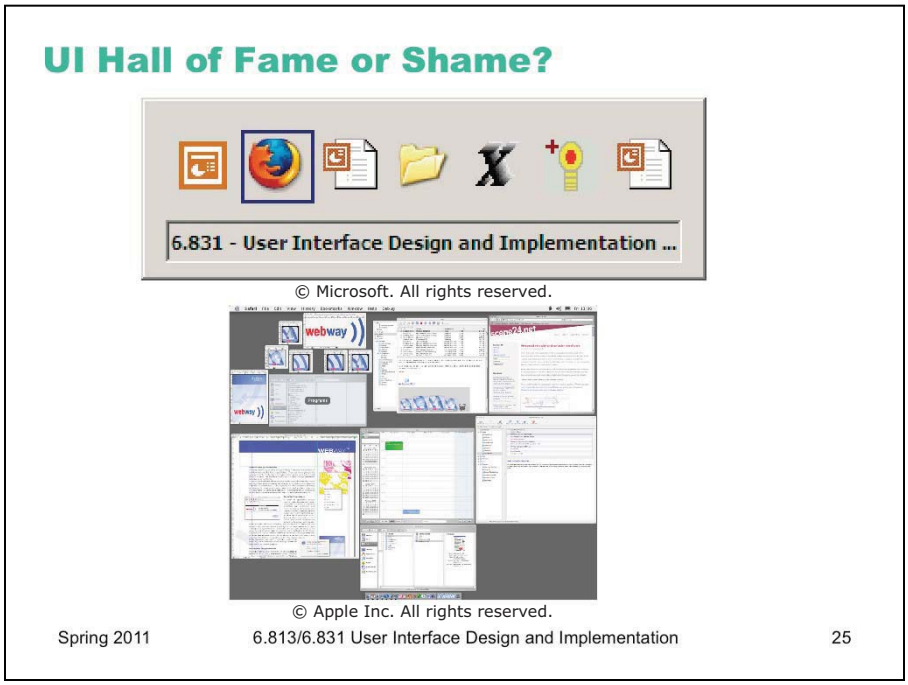
7. full implementation (3 weeks): you'll build a real implementation that you plan to keep.

8. user testing (1 week): you'll test your implementation against users and refine it.

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

Summary

- Models for software development
  - Waterfall model makes sense for low-risk projects
  - Iterative or spiral models are needed when the requirements and design space are unknown or risky
  - UI development is often risky
- User-centered design process
  - Iterative, prototype-driven
  - Early focus on users and tasks
  - Constant evaluation



Next time, our candidates for the Hall of Fame & Shame will be window switching interfaces. We'll start with the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

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

We'll discuss this example in class. Here are a few things to think about:

-how learnable is this interface?

-how visible is it?

-what about efficiency?

- what kinds of errors can you make, and how can you recover from them?

Then we'll talk about **Exposé**, a window-switching interface in Mac OS X. When you push F9 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends Exposé, sending all the other windows back to their old sizes and locations. We'll ask similar questions:

- learnability?

- visibility?

- efficiency?

- errors?

MIT OpenCourseWare
http://ocw.mit.edu

6.831 / 6.813 User Interface Design and Implementation

Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.