Usability

Rob Miller
Fall 2007
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.
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?
Here’s one way it might be redesigned. The templates now fill a list box on the left; selecting a template shows its preview on the right. This interface suffers from none of the problems of its predecessor: list boxes clearly afford selection to new or infrequent users; random access is trivial for frequent users. And no help message is needed.
Here’s another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time look like editable fields (affordance!), but you can’t edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to bring up a dialog box.

This dialog box displays time differently, using 12-hour time (7:17 pm) where the original dialog used 24-hour time (consistency!). Just to increase the confusion, it also adds a third representation, an analog clock face.

So how is the time actually changed? By clicking mouse buttons: clicking the left mouse button increases the minute by 1 (wrapping around from 59 to 0), and clicking the right mouse button increases the hour. Sound familiar? This designer has managed to turn a sophisticated graphical user interface, full of windows, buttons, and widgets, and controlled by a hundred-key keyboard and two-button mouse, into a clock radio!

Perhaps the worst part of this example is that it’s not a result of laziness. Somebody went to a lot of effort to draw that clock face with hands. If only they’d spent some of that time thinking about usability instead.
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. We'll see why later in this lecture.
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.
So user interface development is inherently **risky**. We don’t (yet) have an easy way to predict whether a UI design will succeed.

**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).

Unfortunately, many commercial UI projects inflict iterative design on their paying customers. They design a bad user interface, implement it, 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.

On the other hand, if you keep all your design iterations in-house, you may never release anything! It’s very costly to do every iteration of a design with a high-quality implementation in a language like Java or C++ -- especially if you discover you have to throw away all that code because the design was bad.
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.
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, 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.
Let's start with learnability. This is what many people are thinking about when they use words like "intuitive" or "user-friendly". This example that we saw at the beginning of lecture had serious problems with learnability, because it used the scrollbar in a way that's unfamiliar, inconsistent, and frankly inappropriate.
When you’re designing for learnability, you have to be aware of how people actually learn. You can’t assume that if the interface tells the user something, that the user will immediately learn and remember it. 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. **Just because you've said it, doesn’t mean they know it.** (Incidentally, 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.)
Efficiency concerns how quickly an expert user can operate the system — submitting input or commands, and perceiving and processing the system's output. Note that this is typically not about the performance of the program's algorithms at all – instead, it's about the performance of the I/O channel between the user and the program. A user interface that requires fewer keystrokes to do a task is more efficient. The problem of efficiency is more subtle than just counting keystrokes, however.

Recall the example of Gimp from the start of this lecture. All Gimp menus are accessed from a context menu, which pops up on right-click. You don't have to move your 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! Cascading 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.
Our third goal is error handling. Users make errors; you have to anticipate them, prevent them as much as possible, and deal with them well when they do happen.

Here’s an example of a tricky kind of error created by the keyboard shortcuts in Mozilla Firefox and MIT Webmail. (Thanks to InHan Kang for this example.) The Alt-D shortcut does different things depending on the state you’re in. If you’re browsing any other web site with Firefox, Alt-D puts the keyboard focus on the address bar, so you can type a URL. But if you’re looking at a folder in MIT Webmail, Alt-D deletes the messages you’ve selected. If you’re looking at a message in MIT Webmail, Alt-D normally deletes the message – which at least is consistent with the folder view. But if you’re looking at an already deleted message, then the Delete command is missing – and Alt-D now invokes the Denylist command – which adds the sender of this message to a list of people whose messages get filtered out.

It's easy to see how a user habituated to expect a certain behavior from Alt-D can make serious errors here!

This is an example of a **mode error**. Modes are states of the system 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. If you press Alt-D thinking it will put the focus on the address bar, but it actually deletes an email message, then you’ve made a mode error.
An unfortunately common strategy for error prevention is the confirmation dialog, or “Are you sure?” dialog. It’s not a good approach, and should be used only sparingly, for several reasons:

Confirmation dialogs can substantially reduce the efficiency of the interface. In the example above, a confirmation dialog pops up whenever the user deletes something, forcing the user to make two button presses for every delete, instead of just one. Frequent commands should avoid confirmations.

If a confirmation dialog is frequently seen – for example, every time the Delete button is pressed – then the expert users will learn to expect it, and will start to chunk it as part of the operation. In other words, to delete something, the user will learn to push Delete and then OK, without reading or even thinking about the confirmation dialog! The dialog has then completely lost its effectiveness, serving only to slow down the interface without actually preventing any errors.

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

Clippy was a well-intentioned effort to address learnability problems. 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.
Subjective satisfaction sometimes comes from “eye candy” (beautiful, exciting visual effects). Here’s a neat example of eye candy in Disco, a DVD burner for the Mac:

http://www.youtube.com/watch?v=8JOfiESxac&feature=related

But remember that satisfaction is just one of several dimensions of usability. Asking users “do you like this?” is relevant to usability, but it’s far from the whole story. You should consider learnability, efficiency, and errors as well.
Summary

User interface design is hard
➢ Iterative design is a good approach

Usability
➢ Learnability, efficiency, errors, satisfaction