Joel on Software *painless software management*

[Home](#)

[What's Going On Here?](#)

[Complete Archive](#)

[Translations:](#)
[Brazilian Portuguese](#)
[Bulgarian](#)
[Catalan](#)
[Chinese (Simp)](#)
[Chinese (Trad)](#)
[Danish](#)
[Dutch](#)
[Esperanto](#)
[Farsi](#)
[Filipino](#)
[Finnish](#)
[French](#)
[German](#)
[Greek](#)
[Hebrew](#)
[Hungarian](#)
[Icelandic](#)
[Indonesian](#)
[Italian](#)
[Japanese](#)
[Korean](#)
[Polish](#)
[Portuguese](#)
[Romanian](#)
[Russian](#)
[Spanish](#)
[Swedish](#)
[Tamil](#)
[Turkish](#)
[Ukrainian](#)
...more [soon](#)!

Hosting provided by
[Peer 1 Network](#)

Discuss:
[Joel on Software](#)
[New Yorkers](#)
[.NET Questions](#)
[TechInterview.org](#)
[CityDesk](#)

User Interface Design For Programmers

By Joel Spolsky
Wednesday, October 24, 2001
Printer Friendly Version

Chapter 1: Controlling Your Environment Makes You Happy

Most of the hard core C++ programmers I know *hate* user interface programming. This surprises me, because I find UI programming to be quintessentially easy, straightforward, and fun.

It's *easy* because you usually don't need algorithms more sophisticated than how to center one rectangle in another. It's *straightforward* because when you make a mistake, you immediately see it and can correct it. It's *fun*, because the results of your work are immediately visible. You feel like you are sculpting the program directly.

I think most programmers' fear of UI programming comes from their fear of doing UI *design*. They think that UI design is like graphics design: the mysterious process by which creative, latte-drinking, all-dressed-in-black people with interesting piercings produce cool looking artistic stuff. Programmers see themselves as analytic, logical thinkers: strong at reasoning, weak on artistic judgment. So they think they can't do UI design.

Actually, I’ve found UI design to be quite easy and quite rational. It’s not a mysterious matter that requires a degree from an art school and a penchant for neon-purple hair. There is a rational way to think about user interfaces with some simple, logical rules that you can apply anywhere to improve the interfaces of the programs you work on.

I'm not going to give you "Zen and the Art of UI Design". It's not art, it's not Buddhism, it's just a set of rules. A way of thinking rationally and methodically. This book is designed for programmers. I assume you don't need instructions for *how* to make a menu bar; rather, you need to think about what to put in your menu bar (or whether to have one at all). There is one primary axiom I'll teach you which guides all good UI design, and it's not hard to understand at all.

FogBUGZ

Book Reviews

Email the Author

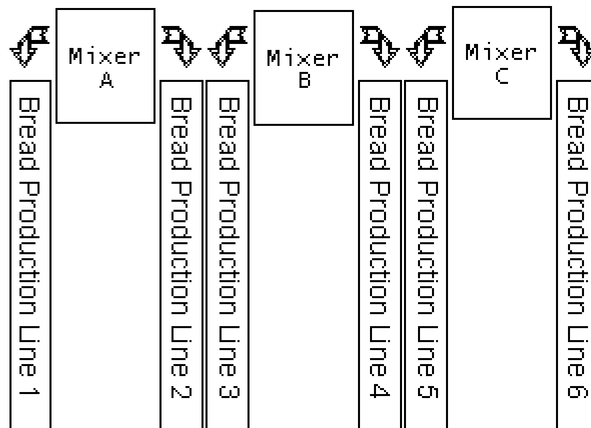
What's CityDesk?

FogBUGZ: Painless Bug Tracking

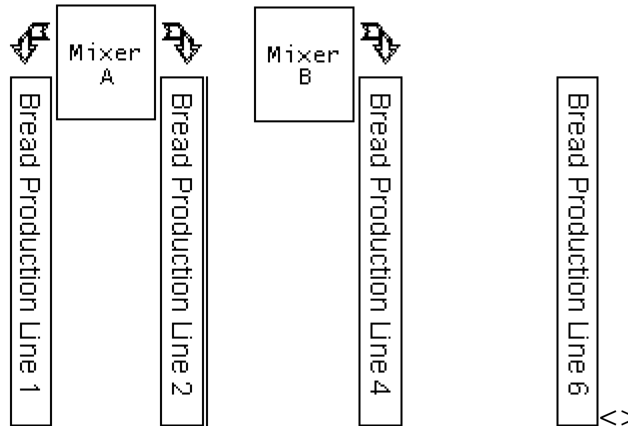
Interview Questions

Search

My first real job was in a big, industrial bakery. The bakery was designed to have six bread production lines. For every two production lines, there was a dough mixer, which produced 180 kg lumps of dough that could be dumped to the left or the right:



Well, this was the design. In reality, Mixer C hadn't been built yet, nor had lines 3 or 5. So the arrangement was:



<>

Alert readers will be wondering, "how did the dough get from Mixer B to production line 6?" Well, that's where Wee Joel came in. My job, if you can believe this, was to stand on the left of Mixer B, then *catch* the giant 180 kg lumps of dough as they flew out of the mixer in a big bathtub-with-wheels, then roll the bathtub over to production line 6, and, using a winch-like device, heave the dough onto line 6. I had to do this once every ten minutes, from about 10 PM until 4 AM.

There were other complications. Line 6 couldn't really handle 180 kg of dough all at once, so I had to slice it with a giant knife into about 10 pieces. I don't even want to go into how absurdly difficult that was.

The first few days, of course, I was terrible at this job. It seemed nearly impossible. Every bone in my body ached. My blisters had blisters. I had aches in places where I didn't know I had places.

At first I just couldn't keep line 6 supplied with dough. Every time they had an interruption in the dough, this caused a big gap on the assembly line. When the gap rolled into the oven, the oven (expending a constant amount of energy over a reduced amount of dough) started to heat up more, which burnt the bread. Sometimes, line 6 would get gummed up and stop production, but the mixer went

right on ahead producing dough for me, and I would run the risk of running out of bathtubs-with-wheels to store the dough in. When this happened, I actually had to clean and oil the floor and dump the dough on the floor to be scraped up later. Not that this would work very well, because if the dough got older than about 30 minutes it would ferment and wouldn't make good bread. If this happened, you had to chop it up into 5 kg pieces and put one piece into the mixture for each future batch.

After a week or so, I got good enough at the routine that I actually had, if I remember correctly, 2 minutes free for every 10 minute dough-cycle to rest. I figured out a precise schedule and learned how to tell the mixer to skip a batch when the production line stopped.

And I started to think about why, as the beer commercial asks, *some days are better than others*.

One day, thinking about this problem, I noticed that one of the bathtubs-with-wheels had pretty lousy wheels that wouldn't turn well. Sometimes this bathtub did not go where I pushed it, and bumped into things. This was a small frustration. Sometimes, as I was pulling the chain to winch up the bathtub, I scraped myself -- just a little bit -- on a splinter of metal on the chain. Another small frustration. Sometimes, as I ran with an empty bathtub to catch a dough emission about to fly out of the mixer, I slipped on a little bit of oil on the floor. Not enough to fall, mind you, just a tiny, small frustration.

Other times, I would have tiny victories. I learned to time the dough production perfectly so that fresh dough would arrive just seconds before the previous batch ran out. This guaranteed the freshest dough and made the best bread. Some of the victories were even tinier: I would spot a tiny blob of dough that had flung off of the mixer and attached itself to the wall, and I would scrape it off with a paint scraper I carried in my back pocket and throw it in the trash. YES! When slicing the dough into pieces, sometimes it just sliced really *nicely* and *easily*. Tiny moments of satisfaction, when I managed to control the world around me, even in the smallest way.

So that's what days were like. A bunch of tiny frustrations, and a bunch of tiny successes. But they *added up*. Even something which seems like a tiny, inconsequential frustration affects your mood. Your emotions don't seem to care about the magnitude of the event, only the quality.

And I started to learn that the days when I was happiest were the days with lots of small successes and few small frustrations.

Years later, when I got to college, I learned about an important theory of psychology called Learned Helplessness, developed by Dr. Martin E. P. Seligman. This theory, backed up by years of research, is that a great deal of depression grows out of a feeling of *helplessness*: the feeling that you cannot control your environment.

The more you feel that you can control your environment, and that the things you do are actually working, the happier you are. When you find yourself frustrated, angry, and upset, it's probably because of something that happened that you could not control: even something small. The space bar on your keyboard is not working well. When you type, some of the words are stuck together. This gets frustrating, because you are pressing the space bar and *nothing is happening.* The key to your front door

doesn't work very well. When you try to turn it, it sticks. Another tiny frustration. These things add up; these are the things that make us unhappy on a day-to-day basis. Even though they seem too petty to dwell on (I mean, there are people *starving* in Africa, for heaven's sake, I can't get upset about *space bars*), nonetheless they change our moods.

Let's pause for a minute and go back to computers.

We're going to invent a typical Windows power user named Pete. When you're thinking about user interfaces, it helps to keep imaginary users in mind. The more realistic the imaginary user is, the better you'll do thinking about how they use your product. Pete is an accountant for a technical publisher who has used Windows for six years at the office and a bit at home. He is fairly competent and technical. He installs his own software; he reads PC Magazine, and he has even programmed some simple Word macros to help the secretaries in his office send invoices. He's getting a cable modem at home. Pete has never used a Macintosh. "They're too expensive," he'll tell you. "You can get a 700 Mhz PC with 128 Meg RAM for the price of..." OK, Pete. We get it.

One day Pete's friend Gena asks him for some computer help. Now, Gena has a Macintosh iBook, because she loves the translucent boxes. When Pete sits down and tries to use the Macintosh, he quickly gets frustrated. "I hate these things," he says. He is, finally, able to help Gena, but he's grumpy and unhappy. "The Macintosh has such a clunky user interface."

Clunky? What's he talking about? *Everybody knows* that the Macintosh has an elegant user interface, right? The very *paradigm* of ease-of-use?

Here's my analysis of this mystery.

On the Macintosh, when you want to move a window, you can grab any edge with the mouse and move it. On Windows, you must grab the title bar. If you try to grab an edge, the window will be reshaped. When Pete was helping Gena, he tried to widen a window by dragging the right edge. Frustratingly, the whole window moved, rather than resizing as he expected.

On Windows, when a message box pops up, you can hit enter *or* the space bar to dismiss the message box. On the Mac, space doesn't work. You usually need to click with the mouse. When Pete got alerts, he tried to dismiss them using the space bar, like he's been doing subconsciously for the last six years. The first time, nothing happened. Without even being aware of it, Pete banged the space bar harder, since he thought that the problem must be that the Mac did not register his tapping the space bar. Actually, it did -- but it didn't care! Eventually he used the mouse. Another tiny frustration.

Pete has also learned to use Alt+F4 to close windows. On the Mac, this actually changes the *volume*. At one point, Pete wanted to click on the Internet Explorer icon on the desktop, which was partially covered by another window. So he hit Alt+F4 to close the window and immediately double-clicked where the icon would have been. The Alt+F4 raised the volume on the computer and didn't close the window, so his double click actually hit the Help button in the toolbar on the window which he wanted closed anyway, which immediately started bringing up a help window, so now, he's got *two* windows open which he has to close.

Another small frustration. But, boy, does it add up. At the end of the day, Pete is grumpy and angry. When he tries to control things, they don't respond. The space bar and the Alt+F4 key "don't work" -- for all intents and purposes, it's as if those keys were broken. The window disobeys him when he tries to make it wider, playing a little prank where it just moves over instead of widening. Bad window. Even if the whole thing is subconscious, the subtle feeling of being out of control translates into helplessness, which translates into unhappiness. "I like my computer," Pete says. "I have it all set up so that it works exactly the way I like it. But these Macs are clunky and hard to use. It's an exercise in frustration. If Apple had been working on MacOS all these years instead of messing around with Newtons, their operating system wouldn't be such a mess."

Right, Pete. We know better. His feelings come *despite* the fact that the Macintosh really is quite easy to use -- for Mac users. It's totally arbitrary which key you press to close a window. The Microsoft programmers, who were, presumably, copying the Mac interface, probably thought that they were adding a cool new feature by letting you resize windows by dragging any edge. The MacOS 8.0 programmers probably thought they were adding a cool new feature when they let you move windows by dragging any edge.

Most flame wars you read about user interface issues focus on the wrong thing. Windows is better because it gives you *more ways* to resize the window. So what? That's missing the point. The point is, does the UI respond to the user in the way in which the user *expected* it to respond? If it didn't, the user is going to feel helpless and out of control, the same way I felt when the wheels of the dough bathtub didn't turn the way I pushed them, and I bumped into a wall. Bonk.

UI is important because it affects the feelings, the emotions, and the mood of your users. If the UI is wrong and the user feels like they can't control your software, they *literally* won't be happy and they'll blame it on your software. If the UI is smart and things work the way the user expected them to work, they will be cheerful as they manage to accomplish small goals. Hey! I ripped a CD! It *just worked! Nice software! Woooooooooo!*

To make people happy, you have to let them feel like they are in control of their environment. To do this, you need to *correctly* interpret their actions. The interface needs to behave in the way they are expecting it to behave.

Thus, the cardinal axiom of all user interface design:

A user interface is well-designed when the program behaves exactly how the user thought it would.

As Hillel said, everything else is commentary. All the other rules of good UI design are just corollaries.

Chapter 2: Figuring Out What They Expected

When a new user sits down to use a program, they do not come with a completely clean slate. They have some expectations of how they think the program is going to

work. If they've used similar software before, they will think it's going to work like that other software. If they've used *any* software before, they are going to think that your software conforms to certain common conventions. They may have intelligent guesses about how the UI is going to work. This is called the *user model*: it is their mental understanding of what the program is doing for them.

The program, too, has a "mental model," only this one is encoded in bits and will be executed faithfully by the CPU. This is called the *program model*, and it is **The Law**. As we learned in Chapter One, if the program model corresponds to the user model, you have a successful user interface.

Let's look at one example. In Microsoft Word (and most word processors), when you put a picture in your document, the picture is actually embedded in the same file as the document itself. You can create the picture, drag it into the document, then *delete the original picture file*, but the picture will still remain in the document.

Now, HTML doesn't let you do this. HTML documents must store their pictures in a separate file. If you take a user who is used to word processors, and doesn't know anything about HTML, and sit them down in front of a nice WYSIWYG HTML editor like FrontPage, they will almost certainly think that the picture is going to be stored in the file. Call this *user model inertia*, if you will.

So we have an unhappy conflict of user model (the picture will be embedded) versus program model (the picture must be in a separate file), and the UI is bound to cause problems.

If you're designing a program like FrontPage, you've just found your first UI problem. You can't really change HTML. Something has to give to bring the program model in line with the user model.

You have two choices. You can try to change the user model. This turns out to be remarkably hard. You could explain things in the manual, but everybody knows that users don't read manuals, and they probably shouldn't have to. You can pop up a little dialog box explaining that the image file won't be embedded, but this has *two* problems: it's annoying to sophisticated users, and users don't read dialog boxes, either (we'll take more about this in Chapter Six).

So, if the mountain won't come to Muhammad ... your best choice is almost always going to be to change the program model, not the user model. Perhaps when they insert the picture, you could make a copy of the picture in a subdirectory beneath the document file, so that at least you can match the user's idea that the picture is copied (and the original can be safely deleted).

How do I know what the user model is?

This turns out to be relatively easy. Just ask them! Pick five random people in your office, or friends, or family, and tell them what your program does in general terms ("it's a program for making web pages"). Then describe the situation: "You've got a web page that you're working on, and a picture file named Picture.JPG. You insert the picture in your web page." Then ask them some questions to try and guess their user model. "Where did the picture go? If you delete Picture.JPG, will the web page still be able to show the picture?"

A friend of mine is working on a photo album application. After you insert your photos, the application shows you a bunch of thumbnails: wee tiny copies of each picture. Now, generating these thumbnails takes a long time, especially if you have a lot of pictures, so he wants to store the thumbnails on the hard drive *somewhere* so that they only have to be generated once. There are a lot of ways he could do this. They could all be stored in one large file called **Thumbnails**. They could all be stored in separate files, in a subdirectory called **Thumbnails**. They might be marked as hidden files in the operating system so that users don't know about them. My friend chose one way of doing it which he thought was the best tradeoff: he stored the thumbnail of each picture **picture.JPG** in a new file named **picture_t.JPG** in the same directory. If you made an album with 30 pictures, when you were done, there were 60 files in the directory including the thumbnail pictures.

You could argue for weeks about the merits and demerits of various schemes of storing the pictures, but as it turns out, there's a more scientific way to do it. Just ask a bunch of users where they think the thumbnails are going to be stored. Of course, many of them won't know or won't care, or they won't have thought about this, but if you ask a lot of people, you'll start to see *some* kind of consensus. The popular choice is the best user model, and it's up to you to make the program model match it.

Next, you have to test your theories. Build a model or prototype of your user interface and give some people tasks to accomplish. As they work through the tasks, ask them what they think is happening. Your goal is to figure out what they expect. If the task is "insert a picture," and you see that they are trying to drag the picture into your program, you'll realize that you better support drag and drop. If they go to the Insert menu, you'll realize that you better have a Picture choice in the Insert menu. If they go to the Font toolbar and replace the word "Times New Roman" with the words "Insert Picture", you've found a relic who hasn't been introduced to GUIs yet and is expecting a command line interface.

How many users do you need to test your interface on? Your instinct may be "the more, the better," which makes sense for scientific experiments. But that instinct is wrong. Almost everybody who does usability testing for a living seems to think that five or six users is enough. After that, you start seeing the same results again and again, and any additional users are just a waste of time.

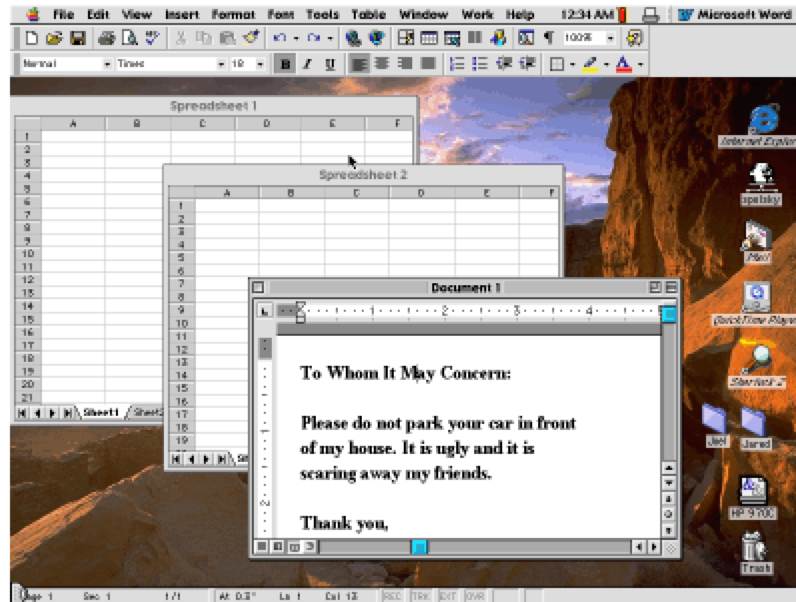
You don't need a formal usability lab, and you don't really need to bring in users "off the street" -- you can do "50 cent usability tests" where you simply grab the next person you see and ask them to try a quick usability test. Make sure you don't spill the beans and tell them how to do things. Ask them to "think out loud" and interview them using open questions to try to discover their mental model.

If your program model is nontrivial, it's probably not the user model.

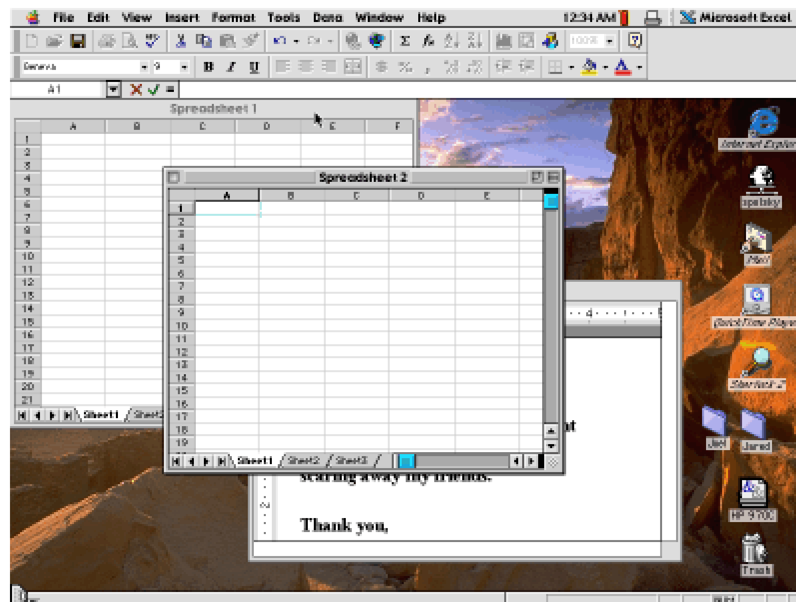
When I was 6 and my dad brought home one of the world's first pocket calculators, an HP-35, he tried to convince me that it had a *computer* inside it. I thought that was unlikely. All the computers on Star Trek were the size of a room and had big reel-to-reel tape recorders. I thought that there was just a clever correlation between the keys on the keypad and the individual elements of the LED display that happened to produce mathematically correct results. (Hey, I was 6).

An important rule of thumb is that user models aren't very complex. When people have to guess how a program is going to work, they tend to guess simple things, rather than complicated things.

Sit down at a Macintosh. Open two Excel spreadsheet files and Word document file. Almost any novice user would guess that the windows were independent. They *look* independent:



The user model says that clicking on Spreadsheet 1 would bring that window to the front. What *really* happens is that Spreadsheet **2** comes to the front, a frustrating surprise for almost anybody:



As it turns out, Microsoft Excel's program model says that "you have these invisible sheets, one for each application, and the windows are 'glued' to those invisible sheets. When you bring Excel to the foreground, all other windows from Excel will move forward, too."

Riiiight. Invisible sheets. What are the chances that the user model included the concept of invisible sheets? Probably about zero. So new users will be surprised by this behavior.

Another example from the world of Microsoft Windows is the Alt+Tab key combination which switches to the "next" window. Most users would probably assume that it simply rotates among all available windows. If you have window A, B, and C, with A active, Alt+Tab should take you to B. Alt+Tab again would take you to C. Actually, what happens is that the second Alt+Tab takes you back to A. The only way to get to C is to *hold down* Alt and press Tab *twice*. It's a nice way to toggle between two applications, but almost nobody figures it out, because it's a slightly more complicated model than the rotate-among-available-windows model.

It's hard enough to make the program model conform to the user model when the models are simple. When the models become complex, it's even more unlikely. So pick the simplest possible model.

Chapter 3: Choices

When you go into a restaurant and you see a sign that says "No Dogs Allowed," you might think that sign is purely proscriptive: Mr. Restaurant doesn't like dogs around, so when he built the restaurant he put up that sign.

If that was *all* that was going on, there would also be a "No Snakes" sign; after all, nobody likes snakes. And a "No Elephants" sign, because they break the chairs when they sit down.

The *real* reason that sign is there is historical: it is a historical marker that indicates that people used to try to bring their dogs into the restaurant.

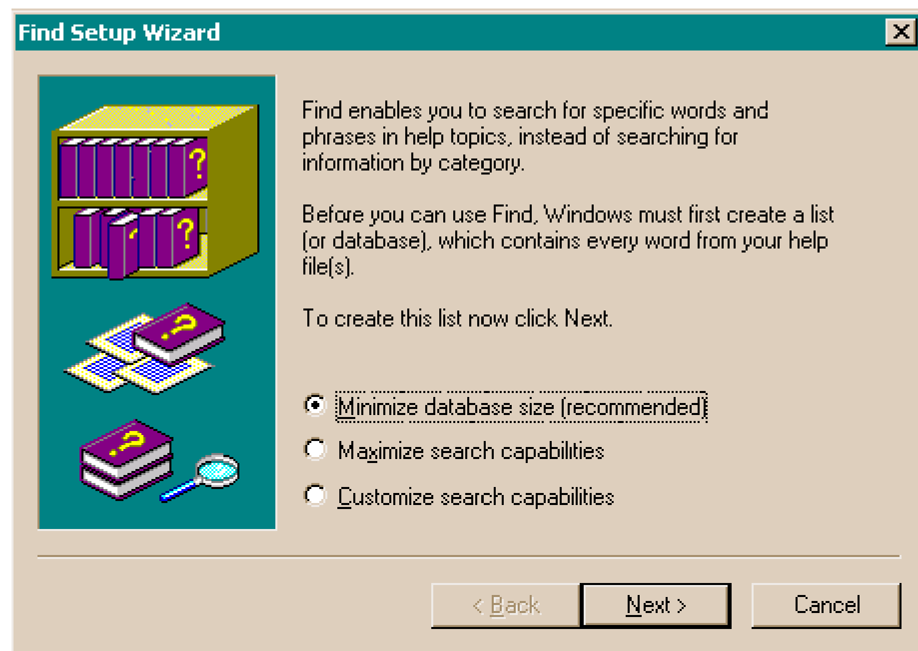
Most prohibitive signs are there because the proprietors of an establishment were sick and tired of people doing X, so they made a sign asking them to please *not*. If you go into one of those fifty year old ma-and-pa diners, like the Yankee Doodle in New Haven, the walls are *covered* with signs saying things like "Please don't put your knapsack on the counter," more anthropological evidence that people used to put their knapsacks on the counter a lot. By the age of the sign you can figure out when knapsacks were popular among local students.

Sometimes they're harder to figure out. "Please do not bring glass bottles into the park" must mean that somebody cut themselves stepping on broken glass while walking barefoot through the grass once, and it's a good bet they sued the city.

Software has a similar archaeological record, too: it's called the Options dialog. Pull up the Tools | Options dialog box and you will see a history of arguments that the software designers had about the design of the product. Should we automatically open the last file that the user was working on? Yes! No! There is a two week debate, nobody wants to hurt anyone's feelings, the programmer puts in an #ifdef in self defense while the designers fight it out. Eventually they just decide to make it an option.

It doesn't even have to be a debate between two people: it can be an internal dilemma. I just *can't decide* if we should optimize the database for size or speed.

Either way, you wind up with things like what is unequivocally the most moronic "wizard" dialog in the history of the Windows operating system. This dialog is so stupid that it deserves some kind of award. A whole new *category* of award. It's the dialog that comes up when you try to find something in Help:



The first problem with this dialog is that it's distracting. You are trying to find help in the help file. You do not, at that particular moment, give a hoot whether the database is small, big, customized, or chocolate-covered. In the meanwhile, this wicked, wicked dialog is giving you little pedantic lectures that it must create a list (or database). There are about three paragraphs there, most of which are completely confusing. There's the painfully awkward phrase "your help file(s)". You see, you may have *one or more* files. As if you **cared** at this point that there could be more than one. As if it made the slightest amount of difference. But the programmer who worked on that dialog was obviously distressed beyond belief at the possibility that there might be more than one help file(s) and it would be incorrect to say help file, now, wouldn't it?

Don't even get me started about how most people who want help are not the kinds of people who understand these kinds of arcana. Or that even advanced users, programmers with PhDs in Computer Science who know *all about* full text indexes, would not be able to figure out what they are really being asked to choose from.

To add insult to injury, this isn't even a dialog... it's a *wizard* (the second page of which just says something like "thank you for submitting yourself to this needless waste of your time," to paraphrase). And it's pretty obvious that the designers had *some* idea as to which choice is best; after all, they've gone to the trouble of recommending one of the choices.

Which brings us to our second major rule of user interface design:

Every time you provide an option, you're asking the user to make a decision.

Asking the user to make a decision isn't *in itself* a bad thing. Freedom of choice can be wonderful. People *love* to order espresso-based beverages at Starbucks because they get to make so many *choices*. Grande-half-caf-skim-mocha-Valencia-with-whip. Extra hot!

The problem comes when you ask them to make a choice that *they don't care about*. In the case of help files, people are looking at the help file because they are having trouble accomplishing something they *really want to accomplish*, like making a birthday invitation. Their birthday invitation task has been unfortunately interrupted because they can't figure out how to print upside down balloons, or whatever, so they are going to the help file. Now, some annoying help-index-engine-programmer at Microsoft with an inflated idea of his own importance to the whole scheme of things has the *audacity*, the *chutzpah*, to interrupt the user *once again* and start teaching the user things about making lists (or databases). This second level of interrupting is completely unrelated to birthday invitations, and it's simply guaranteed to perplex and eventually piss off the user.

And believe you me, users care about a lot less things than you might think. They are using your software to accomplish a task. They care about the task. If it's a graphics program, they probably want to be able to control *every pixel* to the finest level of detail. If it's a tool to build a web site, you can bet that they are obsessive about getting the web site to look exactly the way they want it to look.

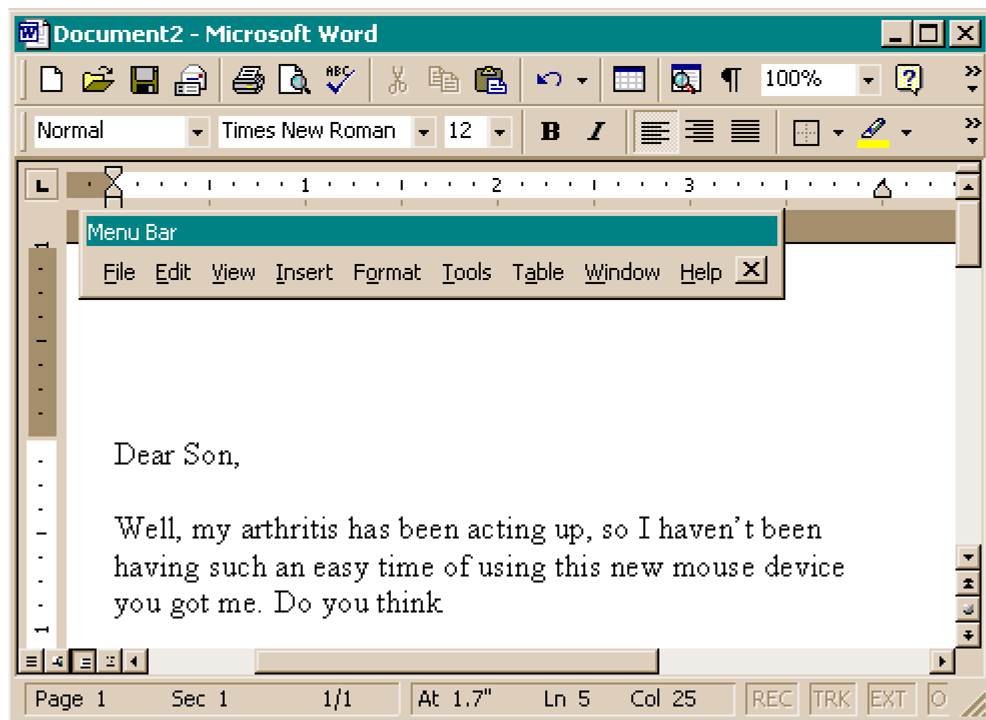
They do *not*, however, care one whit if the program's own toolbar is on the top or the bottom of the window. They don't care how the help file is indexed. They don't care about a lot of things, and it is the designers' responsibility to make these choices for them so that they don't have to. It is the height of arrogance for a software designer to inflict a choice like this on the user simply because the designer couldn't think hard enough to decide which option is really better. (It's even worse when you try to cover up the fact that you're giving the user a difficult choice by converting it to a wizard, as the WinHelp people did. As if the user was a moron who needed to take a little two step mini-course in the choice that they are being offered so that they can make an *educated* decision.)

It has been said that design is the art of *making choices*. When you design a trash can for the corner, you have to make choices between conflicting requirements. It needs to be heavy so it won't blow away. It needs to be light so the trash collector can dump it out. It needs to be large so it can hold a lot of trash. It needs to be small so it doesn't get in peoples' way on the sidewalk. When you are designing, and you try to abdicate your responsibility by forcing the user to decide something, you're probably not doing your job. Someone else will make an easier program that accomplishes the same task with less intrusions, and most users will love it.

When Microsoft Excel 3.0 came out in 1990, it was the first application to sport a new feature called a toolbar. It was a sensible feature, people liked it, and everybody copied it -- to the point that it's unusual to see an application without one any more.

The toolbar was so successful that the Excel team did field research using a special version of Excel which they distributed to a few people; this version kept statistics on what the most frequently used commands were and reported them back to Microsoft. For the next version, they added *another* row of toolbar buttons, this time containing the most frequently used commands. Great.

The trouble was, they never got around to disbanding the toolbar team, who didn't seem to know when to leave good enough alone. They wanted you to be able to *customize* your toolbar. They wanted you to be able to drag the toolbar anywhere on the screen. Then, they started to think about how the menu bar is really just a glorified toolbar with words instead of icons, so they let you drag the *menu bar* anywhere you wanted on the screen, too. Customizability on steroids. Problem: nobody cares! I've never met anyone who wants their menu bar anywhere except at the top of the window. But here's the (bad) joke: if you try to pull down the File menu, and you accidentally grab the menu bar a tiny bit too far to the left, you yank off the whole menu bar, dragging it to the only place you could not possibly want it to be: blocking the document you're working on.

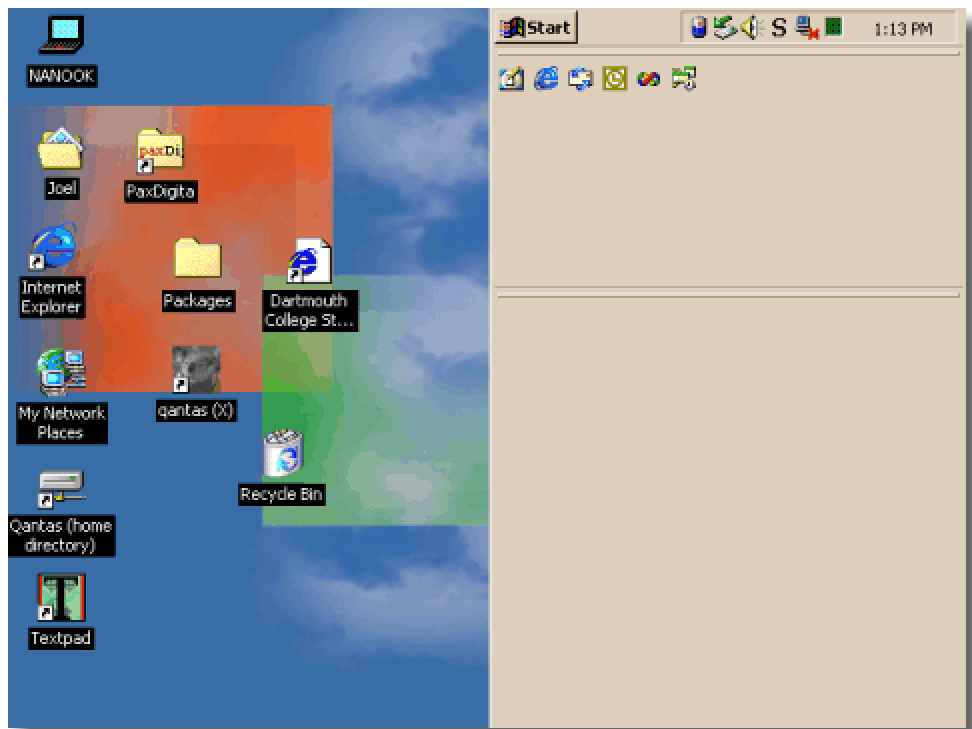


How many times have you seen *that*? And once you've done this by mistake, it's not clear what you did or how to fix it. So here we have an option (moving the menu bar) that nobody wants (ok, maybe 0.1% of all humans want it) but which gets in the way for almost everybody.

One day a friend called me up. She was having trouble sending email. Half the screen was grey, she said.

Half the screen was grey?

It took me five minutes over the phone to figure out what had happened. She had accidentally dragged the Windows toolbar to the right side of the screen, then accidentally widened it:



This is the kind of thing that nobody does *on purpose*. And there are a lot of computer users out there who can't get themselves out of this kind of mess; by definition, when you accidentally reconfigure one of the options in your program, you don't know how to re-reconfigure it. It's sort of shocking how many people uninstall and then reinstall their software when things start behaving wrong, because at least they know how to do that. (They've learned to uninstall first, because otherwise all the broken customizations are likely to just come back).

"But wait!" you say. "It's important to have options for *advanced* users who want to tweak their environments!" In reality, it's not as important as you think. This reminds me of when I tried to switch to a Dvorak keyboard. The trouble was, I don't use *one* computer. I use all kinds of computers. I use other people's computers. I use three computers fairly regularly at home and three at work. I use computers in the test lab at work. The trouble with customizing your environment is that it just doesn't *propagate*, so it's not even worth the trouble.

Most advanced users use several computers regularly; they upgrade their computer every couple of years, they reinstall their operating system every three weeks. It's true that the *first* time they realized you could completely remap the keyboard in Word, they changed everything around to be more to their liking, but as soon as they upgraded to Windows 95 those settings got lost, and they weren't the same at work, and eventually they just stopped reconfiguring things. I've asked a lot of my "power user" friends about this; hardly any of them do any customization other than the bare minimum necessary to make their system behave reasonably.

Every time you provide an option, you're asking the user to make a decision. That means they will have to think about something and decide about it. It's not necessarily a *bad* thing, but, in general, you should always try to minimize the number of decisions that people have to make.

This doesn't mean eliminate *all* choice. There are enough choices that users will

have to make anyway: the way their document will look, the way their web site will behave, or anything else that is integral to the work that the user is doing. In these areas, go crazy: it's great to give people choices: by all means, the more the merrier. And there's another category of choice that people like: the ability to change the visual look of things, without really changing the behavior. Everybody loves WinAmp skins; everybody sets their desktop background to a picture. Since the choice affects the visual look without affecting the way anything functions, and since users are completely free to ignore the choice and get their work done anyway, this is a good use of options.

Chapter 4: Affordances and Metaphors

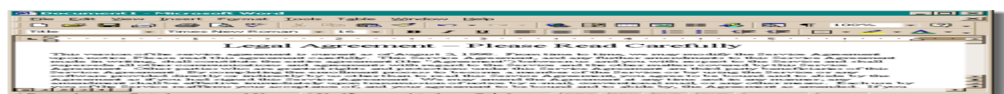
Developing a user interface where the program model matches the user model is not easy. Sometimes, your users might not *have* a concrete expectation of how the program works and what it's supposed to do. In these cases, you are going to have to find ways to give the user clues about how something works. With graphical interfaces, a common way to solve this problem is with *metaphors*. But not all metaphors are created equal, and it's important to understand *why* metaphors work so you know if you've got a good one.

The most famous metaphor is the "desktop metaphor" used in Windows and the Macintosh. You have these little folders with little files in them, which you can drag around. You can drag a file from one folder to another to move it. To the extent that this metaphor works, it's because the little pictures of folders actually remind people of folders, which makes them realize that they can put documents into them.

Here's a screenshot from Kai's Photo Soap. Can you guess how to zoom in?

It's not very hard. The magnifying glass is a real world metaphor. People know what they are supposed to do. And there's no fear that the zoom operation is actually changing the size of the underlying image, since that's not what magnifying glasses do.

A metaphor, even an imperfect one, works a lot better than when you don't have one at all. Can you figure out how to zoom in with Microsoft Word?



Word has two tiny magnifying glasses in their interface, but one of them is on the "Print Preview" button (for some reason), and the other is on the "Document Map" button, whatever that is. The actual way to change the zoom level here is with the dropdown box that is currently showing "100%". There's no attempt at a metaphor, so it's harder for users to guess how to zoom. This is not necessarily a bad thing; zooming is probably not important enough in a word processing application to justify as much screen space as Kai gives it. But it's a safe bet that more Kai users will be able to zoom in than Word users.

My Briefcase

A metaphor, badly chosen, is worse than no metaphor at all. Remember the briefcase from Windows 95? This cute little icon occupied a square inch or so on everybody's desktop for a few years until Microsoft realized that nobody wanted one. And nobody wanted one, because it was a broken metaphor. It was supposed to be a "briefcase", where you put files to take home. But when you took the files home, you still had to put them on a floppy disk. So, do you put them in the briefcase or on a floppy disk? I'm not sure. I don't understand the briefcase. I could never get it to work.

Affordances

Well-designed objects make it clear how they work just by looking at them. Some doors have big metal plates at arm-level. The only thing you can do to a metal plate is push it. In the words of Donald Norman, the plate *affords* pushing. Other doors have big, rounded handles that just make you want to *pull* them. They even imply how they want you to place your hand on the handle. The handle *affords* pulling. It makes you *want* to pull it.

Other objects aren't designed so well and you can't tell what you're supposed to do. The quintessential example is the CD jewel case, which requires you to place your thumbs *just so* and pull in a certain direction. Nothing about the design of the box would indicate how you're supposed to open it. If you don't know the trick, it's very frustrating, because the box just won't open.

The best way to create an affordance is to echo the shape of the human hand in "negative space". Look closely at the (excellent) Kodak DC-290 digital camera, shown here front and back:

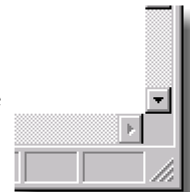
On the front, you can see a big rubber grip which just looks like your right fingers fit there. Even smarter, on the back, in the lower left corner, you can see an indent which looks uncannily like a thumbprint. When you put your left thumb there, your left index finger curls snugly on the front of the camera, between the lens and another rubber nubbin. It provides a kind of comforting feeling you haven't felt since you sucked your thumb (and curled your index finger around your nose).

The Kodak engineers are just trying to persuade you to hold the camera with both hands, in a position which ensures that the camera will be more stable and even keeps stray fingers from blocking the lens by mistake. All this rubber is not functional, its sole purpose is to encourage you to hold the camera correctly.

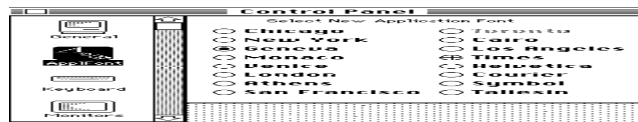
Good computer UI uses affordances, too. About ten years ago, most push buttons went "3D". Using shades of grey, they appear to pop out of the screen. This is not just to look cool: it's important because 3D buttons *afford* pushing. They look like they stick out and they look like the way to operate them is by clicking on them. Unfortunately, many web sites these days (unaware of the value of affordances) would rather have buttons that look *cool* rather than buttons which look *pushable*; as a result, you sometimes have to hunt around to figure out where to click. Look at this web banner:

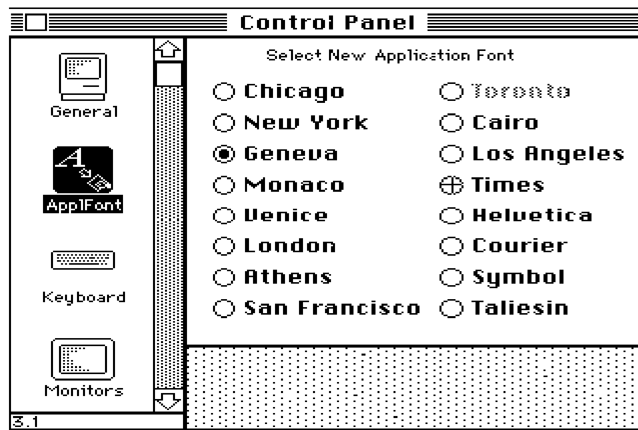
The "Go" and "Login" buttons pop out and *look* like you can click on them. The Site Map and Help buttons don't look so clickable, in fact, they look exactly like the QUOTES label which *isn't* clickable.

About four years ago, many windows started sprouting three little ridges on the bottom right corner which look like a grip. It looks like the kind of thing somebody would put on a slide switch to increase the friction. It *affords* dragging. It just *begs* to be dragged to stretch the window.



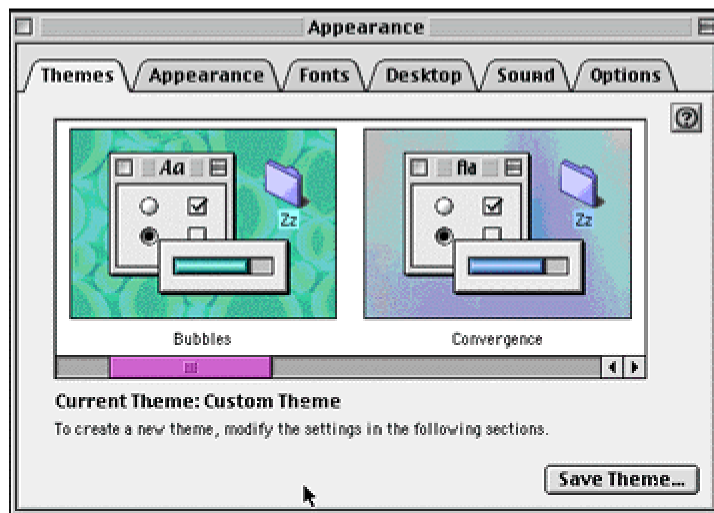
Finally, one of the best examples of affordances is the famous "tabbed dialog". Remember the old Mac control panel?





The idea was that you choose one of the icons from the (scrolling) list on the left. As you click on the icon, the right side of the screen changes. For some reason, this type of indirection was incredibly logical to the programmers who designed it, but many users didn't understand it. Among other things, people rarely figured out how to scroll the list to get more than the first 4 control panels. But more critically, most people just didn't understand that there was a connection between the icons and the dialog. The icons actually look like they are one of the choices.

Starting in about 1992, these interfaces started to disappear, to be replaced with a new invention called tabbed dialogs:



Tabbed dialogs are a great affordance. It's really *obvious* from this picture that you have six tabs; it's really *obvious* which tab you're on, and it's really *obvious* how to switch to a different tab. When Microsoft first usability tested the tabbed dialog interface, usability went up from about 30% (the old Mac way) to 100%. Literally every single testee was able to figure out the tabbed dialogs. Given the remarkable success of this metaphor, and the fact that the code for tabbed dialogs is built into Windows and available practically for free, it's a wonder you still see applications which don't take advantage of them. These applications suffer from actual, measurable, real world usability problems because they refuse to get with the program.

Chapter 5: Consistency and Other Hobgoblins

The main programs in the Microsoft Office suite, Word and Excel, were developed from scratch at Microsoft, but others were bought from outside companies, notably FrontPage (bought from Vermeer) and Visio, bought from Visio. The thing these two programs have in common? They were originally designed to look and feel just like Microsoft Office applications.

The decision to emulate the Office UI wasn't merely to "suck up" to Microsoft or to position the companies for acquisition; indeed, Charles Ferguson, who developed FrontPage, does not hesitate to admit his antipathy for Microsoft; he repeatedly *begged* the Justice department to *do something* about the Redmond Beasties (until he sold his company to them, after which his position became a lot more complicated). In fact Vermeer and Visio seem to have copied the Office UI mainly because it was expedient: it was easier and quicker than reinventing the wheel.

When Mike Mathieu, a group program manager at Microsoft, downloaded FrontPage from Vermeer's web site and tried it out, it worked a whole lot like Word. Since it worked so much like he *expected* a program to work, it was easier to use. And this ease of use gave him a favorable impression of the program right off the bat.

Now, when Microsoft gets a favorable impression of a program right off the bat, they shell out $150 million or so. Your goal is probably more modest; you want your customers to get a favorable impression and shell out maybe $39. But it's the same idea: consistency *causes* ease of use which in turn *causes* good feelings resulting in more money for you.

It's hard to overestimate just how much consistency helps people to learn and use a wide variety of programs. Before GUIs, every program reinvented the very fundamentals of the user interface. Even a simple operation like "exit" which every program *had to* have was completely inconsistent. In those days people made a point of memorizing, at the very least, the exit command of common programs so they could exit and run a program they understood. Emacs fanatics memorized ":q!" (and nothing else) in case they ever found themselves stuck in vi by mistake, while vi users memorized "C-x C-c" (Emacs even has its own way to represent control characters). Over in DOS land, you couldn't even *use* WordPerfect unless you had one of those dorky plastic keyboard templates that reminded you what Alt+Ctrl+F3 did. I just memorized F7 which got you the heck outta there.

Not only that, but small inconsistencies in things like the default typing behavior (overwrite or insert) can drive you *crazy*. I've gotten so used to Ctrl+Z meaning "undo" in Windows applications that when I use Emacs I am constantly minimizing the window (Ctrl+Z) by mistake. (The funny thing is that the very reason Emacs interprets Ctrl+Z as *minimize* is for "consistency" with that terrific user interface, **csh**, the C shell from UNIX.) This is one of those minor frustrations that adds up to a general feeling of unhappiness.

To take an even smaller example, Pico and Emacs both use Ctrl+K to delete lines, but with a *slightly* different behavior that usually mauls my document whenever I find myself in Pico. I'm sure you have a dozen examples of your own.

In the early days of Macintosh, before Microsoft Windows, Apple's evangelists told everyone that the average Mac user used more different programs to get their work

done than the average DOS user. I don't remember the exact numbers, but I believe it was something like 1 or 2 programs for the average DOS user versus *twelve* programs for a Mac user. The reason was that it was so easy to learn a new program on the Mac because they generally worked the same way.

Consistency is a fundamental principle of good UI design, but it's really just a corollary of the axiom "make the program model match the user model", because the user model is likely to reflect the way that users see other programs behaving. If the user has learned that double-clicking text means *select word*, you can show them a program they've never seen before and they will guess that the way to select a word is to double-click it. And now, that program *better* select words when they double click (as opposed to, say, looking the word up in the dictionary), or else you have a usability problem.

If consistency is so *obviously* beneficial, why am I wasting your time and mine evangelizing it? Unhappily, there is a dark force out there that fights against consistency, and that is the natural tendency of designers and programmers to be creative.

Now, I hate to be the one to tell you "don't be creative," but unfortunately, to make a user interface easy to use, you are going to have to channel your creativity into some other area. In most UI decisions, before you design anything from scratch, you absolutely have to look at what other popular programs are doing and emulate that as closely as possible. If you're creating a document editing program of some sort, it better look an awful lot like Microsoft Word, down to the accelerators on the menu items that you have in common. Some of your users will be used to Ctrl+S for save; some of them will be used to Alt+F,S for save, and still others will be used to Alt,F,S (releasing the Alt key) . Another group will look for the floppy disk in the top left area of the program and click it. All four better work, or your users are going to get something that they didn't want.

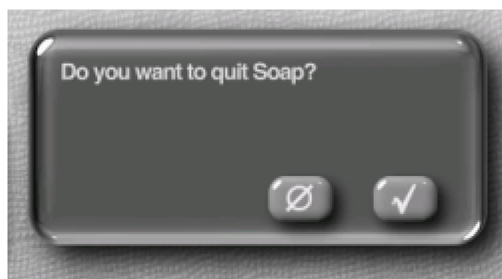
I've seen companies where management prides themselves on doing things *deliberately* differently from Microsoft. "Just because Microsoft does it, doesn't mean it's right," they brag, and then proceed to create a gratuitously different user interface from the one that people are used to. Before you start chanting the mantra that "just because Microsoft does it, doesn't mean it's right," please consider two things:

1. Even if it's not right, if Microsoft is doing it in a popular program like Word, Excel, Windows, or Internet Explorer, then millions of people are going to *think* that it's right, or at least, fairly standard, and they are going to assume that your program works the same way. Even if you think (as the Netscape 6.0 engineers clearly do) that Alt+Left is not a good shortcut key for "Back", there are literally millions of people out there who will try to use Alt+Left to go back, and if you refuse to do it on some general religious principle that Bill Gates is the evil smurf arch-nemesis Gargamel, then you are just gratuitously ruining your program so that you can feel smug and self-satisfied, and your users will not thank you for it.
2. And don't be so sure it's not right. Microsoft spends more money on usability testing than you do, they keep detailed statistics based on millions of tech support phone calls, and there's a darn good chance that they did it that way because more people can figure out how to use it that way.

To create a good program with a usable user interface, you're going to have to leave your religion at the door, thank you. Microsoft may not be the only company to copy: if you're making an online bookstore, you should probably make sure that your web site is at least semantically the same as Amazon. Amazon keeps your shopping cart around for 90 days. You might think that you are extra-smart and empty the cart after 24 hours. If you do this, there will be Amazon customers who put stuff in your shopping cart and come back two weeks later expecting it to still be there. When it's gone, you've lost a customer.

If you're making a high end photo editor for graphics professionals, I assure you that 90% of your users are going to know Adobe Photoshop, so you better behave a heck of a lot like Photoshop in the areas where your program overlaps. If you don't, people are going to say that your program is hard to use, even if *you* think it's easier to use than Photoshop, because it's not behaving the way *they* expect it to.

There is another popular tendency to reinvent the common controls that come with Windows. Don't even get me started about Netscape 6. There was a time when you could tell the programs that were compiled with Borland's C++ compiler because they used big fat OK buttons with giant green checkboxes. This wasn't nearly as bad as Kai's Photo Soap:



Fine, so, it's stunningly beautiful, but the O with a line through it (which actually means "no") reminds me of "OK," and the standard on Windows is to have OK on the left, so I wind up hitting the wrong button a lot. The only benefit to having funny symbols instead of "OK" and "Cancel" like everyone else is that you get to show off how *creative* you are. If people make mistakes because of Kai's creativity, well, that's just the price they have to pay for being in the presence of an *artist*. (Another problem with this "dialog" is that it doesn't have a standard title bar which can be used to move the dialog around on the screen. So if the dialog gets in the way of something you want to see in order to answer the question in the dialog, you are out of luck.)

Now, there's a lot to be gained by having a slick, cool-looking user interface. Good graphical design like Kai is pleasing and will attract people to your program. The trick is to do it *without* breaking the rules. You can change the visual look of dialogs, a bit, but don't break the functionality.

When the first version of Juno was written, it had the standard log on dialog that prompted you for a user name and a password. After you entered the user name, you were supposed to press TAB to go to the password field and type in a password.

Now, this distracted one of the programming managers at Juno, who had a lot more experience with UNIX than with Windows, so he was used to typing user name, then pressing ENTER to jump to the password field (instead of TAB). Now, when you're

writing a program targeted at non-expert Windows users, a UNIX programmer is probably *not* the ideal example of a typical user, but this manager was very insistent that the enter key should move to the next field instead of doing the Windows-standard "OK" thing. "Just because Microsoft does it, doesn't mean it's right," he chirped.

So the programmers spent a really remarkable amount of time writing some amazingly complicated dialog box handling code to work around the default behavior of Windows. (Being inconsistent is almost always *more* work than just acting like your platform expects you to act). This code was a big maintenance nightmare; it didn't port so well when we moved from 16-bit to 32-bit Windows. It didn't do what people expected. And as new programmers joined the team, they didn't understand why there was this strange subclass for dialogs.

An awful lot of programmers have tried to reimplement various common Windows controls, from buttons to scrollbars to toolbars and menu bars (the Microsoft Office team's favorite thing to reimplement). Netscape 6.0 goes so far as to reimplement every single common Windows control. This usually has some unforeseen bad effects. The best example is with the edit box. If you reimplement the edit box, there are an awful lot of utilities that you don't even know about (like Chinese language editing add-ins, and bidirectional versions of Windows that support right-to-left text) that are going to stop working because they don't recognize your non-standard edit box. Some of the reviewers of the preview release of Netscape 6.0 noticed that the URL box, using a non-standard Netscape edit control, does not support common edit control features like right clicking to get a context menu.

When you find yourself arguing with a anti-Microsoft fundamentalist or a creative graphic designer about consistency, they're apt to quote Emerson incorrectly: "Consistency is the hobgoblin of little minds..." The real quote is "A *foolish* consistency is the hobgoblin of little minds." Good UI designers use consistency intelligently, and, though it may not show off their creativity as well, in the long run it makes users happier.

Chapter 6: Designing for People Who Have Better Things To Do With Their Lives

When you design user interfaces, it's a good idea to keep two principles in mind:

1. Users don't have the manual, and if they did, they wouldn't read it.
2. In fact, users can't read anything, and if they could, they wouldn't want to.

These are not, strictly speaking, *facts*, but you should act as if they are facts, for it will make your program easier and friendlier. Designing with these ideas in mind is called *respecting the user*, which means, not having much respect for the user. Confused? Let me explain.

What does it mean to make something *easy to use*? One way to measure this is to see what percentage of real-world users are able to complete tasks in a given amount of time. For example, suppose the goal of your program is to allow people to convert digital camera photos into a web photo album. If you sit down a group of average users with your program and ask them all to complete this task, then the more *usable* your program is, the higher the percentage of users that will be able to successfully create a web photo album. To be scientific about it, imagine 100 real

world users. They are not necessarily familiar with computers. They have many diverse talents, but some of them distinctly do *not* have talents in the computer area. Some of them are being distracted while they try to use your program. The phone is ringing. WHAT? The baby is crying. WHAT? And the cat keeps jumping on the desk and batting around the mouse. I CAN'T HEAR YOU!

Now, even without going through with this experiment, I can state with some confidence that some of the users will simply fail to complete the task, or will take an extraordinary amount of time doing it. I don't mean to say that these users are *stupid*. Quite the contrary, they are probably highly intelligent, or maybe they are accomplished athletes, but vis-à-vis *your program*, they are just not applying all of their motor skills and brain cells to the usage of your program. You're only getting about 30% of their attention, so you have to make do with a user who, from inside the computer, does not appear to be playing with a full deck.

Users Don't Read the Manual.

First of all, they actually don't *have* the manual. There may not *be* a manual. If there is one, the user might not have it, for all kinds of logical reasons: they're on the plane; they are using a downloaded demo version from your web site; they are at home and the manual is at work; their IS department never *gave* them the manual. Even if they have the manual, frankly, they are simply not going to read it unless they absolutely have no other choice. With *very* few exceptions, users will not cuddle up with your manual and read it through before they begin to use your software. In general, your users are trying to get something *done*, and they see reading the manual as a waste of time, or at the very least, as a distraction that keeps them from getting their task done.

The very fact that you're reading this book puts you in an elite group of highly literate people. Yes, I know, people who use computers are by and large *able* to read, but I guarantee you that a good percentage of them will find reading to be a chore. The language in which the manual is written may not be their first language, and they may not be totally fluent. They may be kids! They can decipher the manual if they really *must*, but they sure ain't gonna read it if they don't have to. Users do just-in-time manual reading, on a strictly need-to-know basis.

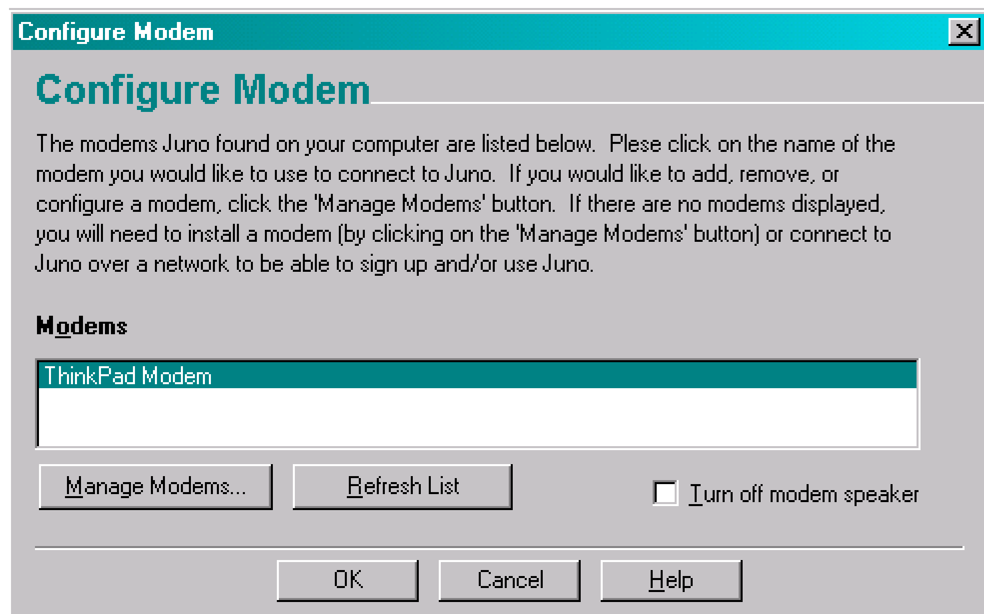
The upshot of all this is that you probably have no choice but to design your software so that it does not need a manual in the first place. The only exception I can think of is if your users do not have any *domain knowledge* -- they don't really understand what the program is intended to do, but they know that they better learn. A great example of this is Intuit's immensely popular small-business accounting program QuickBooks. Many of the people who use this program are small business owners who simply have no idea what's involved in accounting. The manual for QuickBooks assumes this and assumes that it will have to teach people basic accounting principles. There's no other way to do it. Still, if you do know accounting, QuickBooks is easy to use without the manual.

In fact, users don't read *anything*.

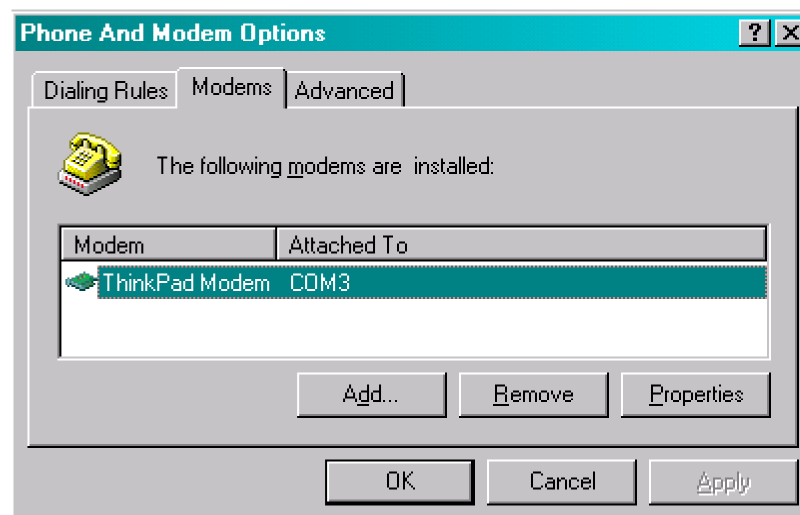
This may sound a little harsh, but you'll see, when you do usability tests, that there are quite a few users who simply do not read words that you put on the screen. If you

pop up an error box of any sort, they simply will not read it. This may be disconcerting to you as a programmer, because you imagine yourself as conducting a *dialog* with the user. Hey, user! You can't open that file, we don't support that file format! Still, experience shows that the more words you put on that dialog box, the fewer people will actually read it.

The fact that users do not read the manual leads many software designers to assume that they are going to have to educate users by describing things as they go along. You see this all over the place in programs. In principle, it's OK, but in reality, people's aversion to reading means that this will almost always get you in trouble. Experienced UI designers literally try to minimize the number of words on dialogs to increase the chances that they will get read. When I worked on Juno, the UI people understood this principle and tried to write short, clear, simple text. Sadly, the CEO of the company had been an English major at an Ivy League college; he had no training in UI design or software engineering, but he sure *thought* he was a good editor of prose. So he vetoed the wording done by the professional UI designers and added lots of his own verbiage. A typical dialog in Juno looks like this:



Compare that to the equivalent dialog from Windows:

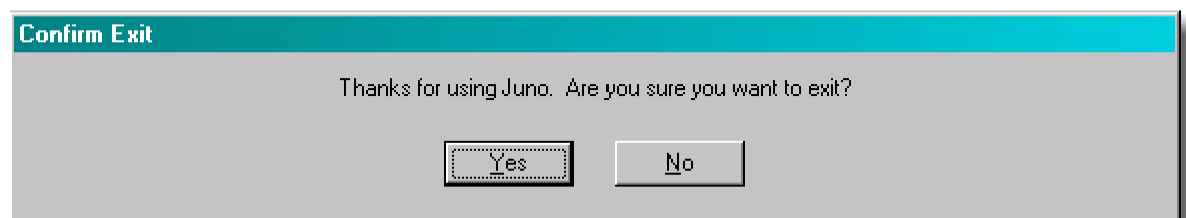


Intuitively, you might guess that the Juno version, with 80 words of instructions, would be "superior" (i.e., easier to use) than the Windows version, with 5 words of instructions. In reality, when you run a usability test on this kind of thing, you'll find that

- advanced users skip over the instructions. They assume they know how to use things and don't have time to read complicated instructions
- most novice users skip over the instructions. They don't like reading too much and hope that the defaults will be OK
- the remaining novice users who do, earnestly, try to read the instructions (some of whom are only reading them because it's a usability test and they feel obliged) are often confused by the sheer number of words and concepts. So even if they were pretty confident that they would be able to use the dialog when it first came up, the instructions actually *confused them even more*.

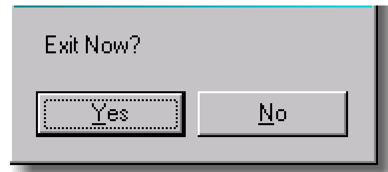
Now, Juno was obviously micro-managed beyond all reason. More to the point, if you're an English major from Columbia, then you are in a whole different *league* of literacy than the average Joe, and you should be very careful about wording dialogs that look helpful to you. Shorten it, dumb it down, simplify, get rid of the complicated clauses in parentheses, and usability test. But do *not* write things that look like Ivy League faculty memos. Even adding the word "please" to a dialog, which may seem helpful and polite, is going to slow people down: the increased bulk of the wording is going to reduce, by some measurable percentage, the number of people who read the text.

Another important point is that **many people are intimidated by computers**. You probably know this, right? But you may not realize the implications of this. I was watching a friend try to exit Juno. For some reason she was having quite a bit of trouble. I noticed that when you try to exit Juno, the following dialog pops up:



She was hitting **No**, and then she was kind of surprised that Juno hadn't exited. The very fact that Juno was questioning her choice made her immediately assume that she was doing something wrong. Usually, when programs ask you to confirm a command, it's because you're about to do something which you might regret. She had assumed that if *the computer* was questioning her judgment, then *the computer* must have been right, because, after all, computers are *computers* where as she was merely a *human*, so she hit "No."

Is it too much to ask people to read 11 lousy words? Well, apparently. First of all, since exiting Juno has no deleterious effects, Juno should have just exited without prompting for confirmation, like every other GUI program in existence. But even if you are *convinced* that it is *crucial* that people confirm before exiting, you could do it in two words instead of 11:



Without the completely unnecessary "thank you" and the remorse-inspiring "are you *sure?*", this dialog is a lot less likely to cause problems. Users will certainly read the two words, say "um, duh?" to the program, and pound the Yes key.

Sure, the Juno Exit Confirmation dialog trips up a *few* people, you say, but is it *that* big a deal? Everyone will *eventually* manage to get out of the program. But herein lies the difference between a program which is *possible* to use versus a program which is *easy* to use. Even smart, experienced, advanced users will appreciate things that you do to make it easy for the distracted, inexperienced, beginner users. Hotel bathtubs have big grab bars. They're just there to help disabled people, but everybody uses them anyway to get out of the bathtub. They make life easier even for the physically fit.

In the next chapter, I'll talk a bit about the mouse. Just like users don't/won't/can't read, some are not very good at using the mouse, so you have to accommodate them.

Chapter 7: Designing for People Who Have Better Things To Do With Their Lives, Part Two

When the Macintosh was new, Bruce "Tog" Tognazzini wrote a column in Apple's developer magazine on UI. In his column, people wrote in with lots of interesting UI design problems, which he discussed. These columns continue to this day on his web site. They've also been collected and embellished in a couple of great books, like Tog on Software Design, which is a lot of fun and a great introduction to UI design. (Tog on Interface was even better, but it's out of print.)

Tog invented the concept of the *mile high menu bar* to explain why the menu bar on the Macintosh, which is always glued to the top of the physical screen, is so much easier to use than menu bars on Windows, which appear *inside* each application window. When you want to point to the File menu on Windows, you have a *target* about half an inch wide and a quarter of an inch high to acquire. You must move and position the mouse fairly precisely in both the vertical and the horizontal dimensions.

But on a Macintosh, you can slam the mouse up to the top of the screen, without regard to how high you slam it, and it will stop at the physical edge of the screen - the correct vertical position for using the menu. So, effectively, you have a target that is still half an inch wide, but a mile high. Now you only need to worry about positioning the cursor horizontally, not vertically, so the task of clicking on a menu item is that much easier.

Based on this principle, Tog has a pop quiz: what are the five spots on the screen that are easiest to acquire (point to) with the mouse? The answer: all four corners of the screen (where you can literally slam the mouse over there in one fell swoop without any pointing at all), plus, the current position of the mouse, because it's already there.

The principle of the mile-high menu bar is fairly well known, but it must not be entirely obvious, because the Windows 95 team missed the point *completely* with the Start push button, sitting *almost* in the bottom left corner of the screen, but not *exactly*. In fact, it's about 2 pixels away from the bottom and 2 pixels from the left of the screen. So, for the sake of a couple of pixels, Microsoft literally *"snatches defeat from the jaws of victory"*, Tog writes, and makes it that much harder to acquire the start button. It could have been a mile square, absolutely trivial to hit with the mouse. For the sake of something, I don't know what, it's not. God help us.

In the previous chapter, we talked about how users hate reading, and will avoid it unless they absolutely cannot accomplish their task. Similarly:

Users can't control the mouse very well.

I don't mean this literally. What I mean is, you should design your program so that it does not require a tremendous amount of mouse-agility to use it right. Top six reasons:

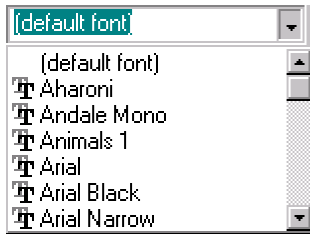
1. Sometimes people are using sub-optimal pointing devices, like trackballs, trackpads, and the little red thingy on a ThinkPad, which are harder to control than true mice.
2. Sometimes people are using mice under bad conditions: a crowded desk; a dirty trackball making the mouse skip; or the mouse itself is a $5 clone which just doesn't track right.
3. Some people are new to computers and have not yet developed the motor skills to use mice accurately.
4. Some people literally will never have the motor skills to use mice precisely, and never will. They may have arthritis, tremors, carpal tunnel; they may be very young or very old; or any other number of disabilities.
5. Many people find that it is extremely difficult to double-click without slightly moving the mouse. As a result they often drag things around on their screen when they mean to be launching applications. You can tell these people because their desktops are a mess because half the time they try to launch something, they wind up moving it instead.
6. Even in the best of situations, using the mouse a lot *feels slow* to people. If you force people to perform a multi-step operation using the mouse, they may feel like they are being stalled which in turn makes the UI feel unresponsive, which, as you should know by now, makes them unhappy.

In ye olden days when I worked on Excel, laptops didn't come with pointing devices built in, so Microsoft made a clip-on trackball that clipped to the side of the keyboard. Now, a mouse is controlled with the wrist and most of the fingers. This is much like writing, and you probably developed very accurate motor skills for writing in elementary school. But a trackball is controlled entirely with the thumb. As a result, it's much harder to control a trackball to the same degree of accuracy as a mouse. Most people find that they can control a mouse to within one or two pixels, but can only control a trackball to within 3 or 4 pixels. On the Excel team, I always urged people to try out their new UIs with the trackball, instead of only with a mouse, to see how it would feel to people who are not able to get the mouse to go exactly where they want it.

One of the UI elements which bothers me the most is the dropdown combo list box.

That's the one that looks like this:

When you click on the down arrow, it expands:



Think about how many detailed mouse clicks it's going to take to choose, say, Times New Roman. First, you have to click on the down arrow. Then, using the scroll bar, you have to carefully scroll until Times New Roman is in view. Many of these dropdowns are carelessly designed to show only two or three items at a time, so this scrolling is none too easy, especially if you have a lot of fonts. It involves either carefully dragging the thumb (with such a small range of movement, it's probably unlikely that this will work), or clicking repeatedly on the second down arrow, or trying to click in the area between the thumb and the down area -- which will eventually stop working when the thumb gets low enough, annoying you even further. Finally, if you do manage to get Times New Roman into view, you have to click on it. If you miss, you get to start all over again. Now multiply by 10, if, say, you want to use a fancy font for the first letter in each of your chapters, and you're *really* unhappy.

The poxy combo dropdown control is even more annoying because there's such an easy solution: just make the dropdown long enough to contain all of the options. 90% of the combo boxes out there don't even use all available space to drop down, which is a *sin*. If there is not enough room between the main edit box and the bottom of the screen, the dropdown should grow *up* until it fits all the items, even if it has to go all the way from the top of the physical screen to the bottom of the physical screen. And then, if there are still more items than fit, let the combo scroll automatically as the mouse approaches the edge, rather than requiring the poor user to mess with a teensy weensy scrollbar.

Furthermore, don't make me click on the little tiny arrow to the right of the edit box before you pop up the combo: let me click *anywhere* on the combo box. This expands the click target about tenfold and makes it that much easier to acquire the target with the mouse pointer.

Let's look at another problem with mousing: edit boxes. You may have noticed that almost every edit box on the Macintosh uses a `fat, wide, bold font` called Chicago which looks kind of ugly and distresses graphic designers to no end. Graphic designers (unlike UI designers) have been taught that thin, variable spaced fonts are more gracious, look better, and are easier to read. All this is true. But graphic designers learned their skills on *paper*, not on the screen. When you need to *edit* text, monospace has a major advantage over variable spaced fonts: it's easier to see and select narrow letters like "l" and "i". I learned this lesson after watching a sixty year old man in a usability test painfully trying to edit the name of his street, which was something like Fillmore Street. We were using 8 point Arial, so the edit box looked like this:

Notice that the I and the Ls are literally *one pixel wide*. The difference between a lower case I and a lower case L is literally *one pixel.* (Similarly, it is almost impossible to see the difference between "RN" and "M" in lower case, so this edit box might actually say Fillrnore.)

There are very few people who would notice if they mistyped Flilmore or Fiilmore or Fillrnore, and even if they did, they would have a *heck* of a time trying to use the mouse to select the offending letter and correct it. In fact, they would even have a hard time using the blinking cursor, which is two pixels wide, to select a single letter. Look how much easier it would have been if we had used a fat font (shown here with Courier Bold)

Fine, OK, so it takes up more space and doesn't look as cool to your graphic designers. Deal with it! It's much easier to use; it even *feels* better to use because as the user types, they get sharp, clear text, and it's so much easier to edit.

Here's a common programmer thought pattern: there are only three numbers: 0, 1, and *n*. If *n* is allowed, all *n*'s are equally likely. This thought pattern comes from the belief (probably true) that you shouldn't have any numeric constants in your code except for 0 and 1. (Constants other than 0 and 1 are referred to as "magic numbers". I don't even want to go into the gestalt of *that*.)

Thus, for example, programmers tend to think that if your program allows you to open multiple documents, it must allow you to open *infinitely* many documents (as memory allows), or at least 2^32, the only magic number programmers concede. A programmer would tend to look with disdain on a program which limited you to 20 open documents. What's 20? Why 20? It's not even a power of 2!

Another implication of *all n's are equally likely* is that programmers have tended to think that if users are allowed to resize and move windows, they should have *complete* flexibility over where these windows go, right down to the last pixel. After all, positioning a window 2 pixels from the top of the screen is "equally likely" as positioning a window *exactly* at the top of the screen.

But it's not true. As it turns out, there are lots of good reasons why you might want a window exactly at the top of the screen (it maximizes screen real estate), but there aren't any reasons to leave 2 pixels between the top of the screen and the top of the window. So, in reality, 0 is much more likely than 2.

The programmers over at Nullsoft, creators of WinAmp, managed somehow to avoid the programmer-think that has imprisoned the rest of us for a decade. WinAmp has a great feature. When you start to drag the window *near* the edge of the screen, coming within a few pixels, it automatically *snaps* to the edge of the screen perfectly. Which is probably exactly what you wanted, since 0 is so much more likely than 2. (The Juno main window has a similar feature: it's the only application I've ever seen that is "locked in a box" on the screen and cannot be dragged beyond the edge.)

You lose a little bit of flexibility, but in exchange, you get a user interface that

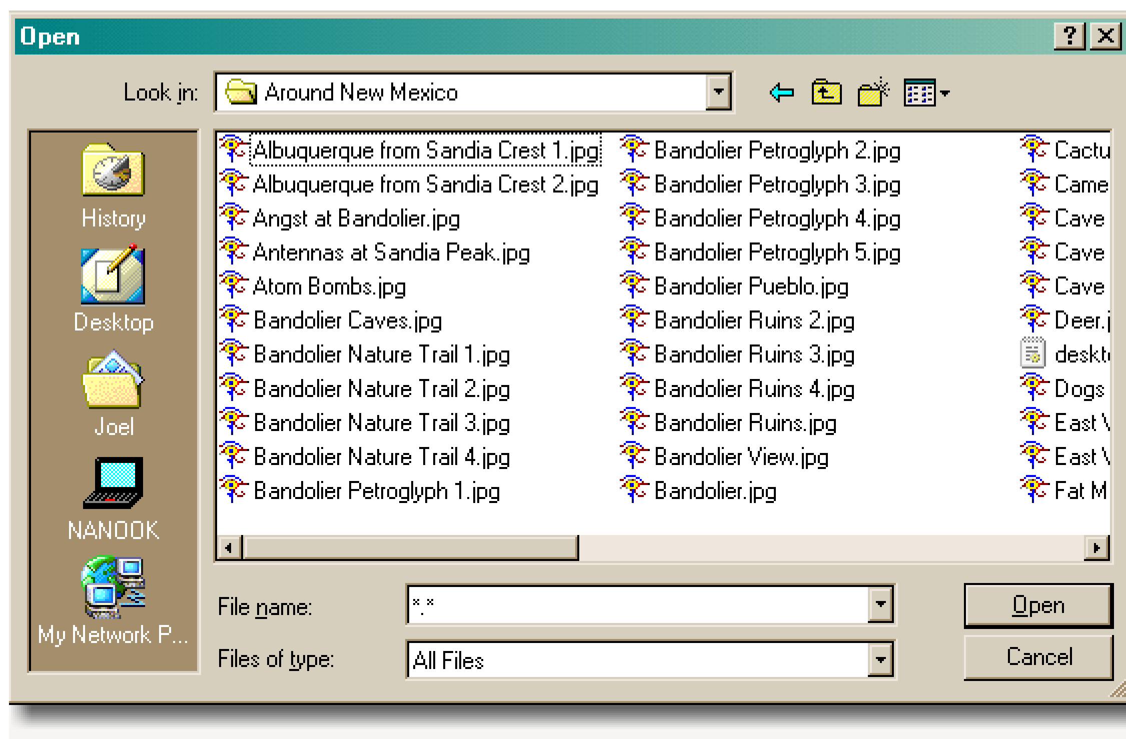
recognizes that controlling the mouse precisely is hard, so why should you have to? This innovation (which every program could use) eases the burden of window management in an intelligent way. Look closely at your user interface, and give us all a break. Pretend that we are gorillas, or maybe smart orangutans, and we really have trouble with the mouse.

Chapter 8: Designing for People Who Have Better Things To Do With Their Lives, Part Three

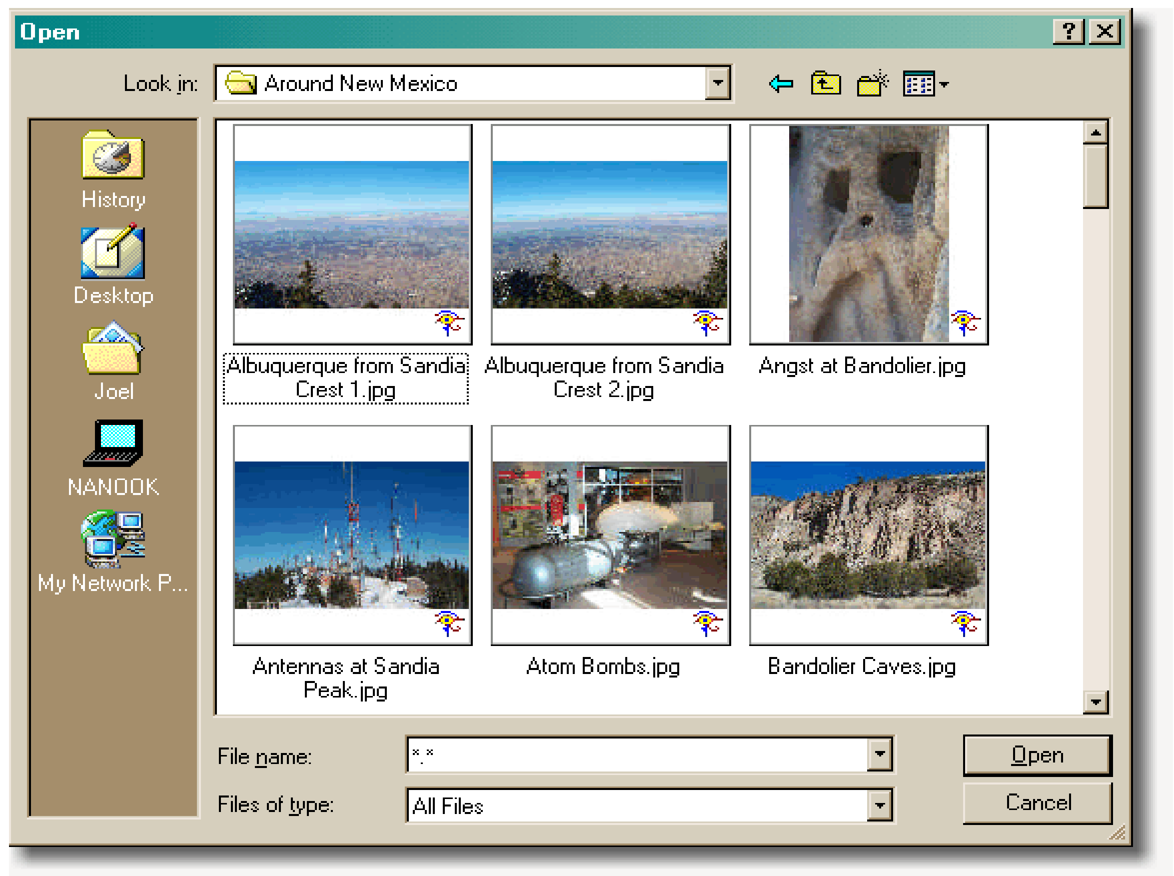
One of the early principles of GUI interfaces was that you shouldn't ask people to remember things that the computer could remember. The classic example is the Open File dialog box, which shows people a list of files rather than asking them to recall and type the exact file name. People remember things a lot better when they are given some clues, and they'd always rather choose something from a list than have to recall it from memory.

Another example is the menus themselves. Historically, providing a complete menu of available commands replaced the old command-line interfaces, where you had to memorize the commands you wanted to use. And this is, fundamentally, the reason why command line interfaces are just **not** better than GUI interfaces, no matter what your UNIX friends tell you. Using a command line interface is like having to learn Korean to order food in a Seoul branch of McDonalds. Using a menu based interface is like being able to point to the food you want and nod your head vigorously: it conveys the same information with no learning curve.

Consider the file selection process in a typical graphics program:



Luckily, Windows 98 introduced thumbnail support, so you can see the files like this:



This makes it significantly easier to open the file you want; it doesn't even take the mental effort to map words onto pictures.

You can see the minimum-memory principle at work in features like auto completion, too. Even if you need to type something, some programs make educated guesses about what you're about to type:

Book1

	A	B	C	D	E
1	Name	Age	Sex		
2	Joel	23	Male		
3	Jenine	29	Female		
4	Micah	39	Male		
5					
6					
7					

Sheet1 Sheet2 Sheet3

In this example, as soon as you type "M", Excel guesses that you are likely to be typing Male, because you've typed Male before in this column, and proposes that as the auto completion. But the "ale" is *preselected* so that if you didn't *mean* to type Male, you can keep typing (perhaps "ystery") and overwrite Excel's guess with no lost effort.

Microsoft Word gets a little bit carried away guessing what you are about to type, as anybody that has ever used that product during the merry month of May has

discovered:

May 8, 2000

I think I may

Designing For People Who Have Better Things To Do With Their Lives, Redux

In the preceding chapters, I've brought up three principles:

- Users don't read stuff (chapter 6)
- Users can't use the mouse (chapter 7)
- Users can't remember anything

You might be starting to get the impression that I think that users are dolts. It's not true. Disrespecting your users is how arrogant software like Microsoft Bob gets created (and dumped in the trash bin), and nobody is very happy.

On the other hand, there is a much worse kind of arrogance in software design: the arrogant assumption that "my software is so damn cool, people are just going to have to warp their brains around it." This kind of *chutzpah* is pretty common in the free software world. Hey, Linux is free! If you're not smart enough to decipher it, you don't deserve to be using it!

Human aptitude tends towards the bell curve. Maybe 98% of your customers are smart enough to use a television set. About 70% of them can use Windows. 15% can use Linux. 1% can program. But only 0.1% of them can program in a language like C++. And only 0.01% of them can figure out Microsoft ATL programming. (And all of them, without exception, have beards and glasses.)

The effect of this sharp drop-off is that whenever you "lower the bar" by even a small amount, making your program, say, 10% easier to use, you *dramatically* increase the number of people who can use it, say, by 50%.

So, I don't really believe that people are dolts, but I think that if you constantly try to design your program so that it's easy enough for dolts to use, you are going to make a popular, easy to use program that people like. And you will be surprised by how what seem like small usability improvements translate into lots more customers.

One good way to evaluate the usability of a program or dialog you've never seen before is to act a little stupid. Don't read the words on the dialog. Make random assumptions about what things do without verifying. Try to use the mouse with just one finger. Make lots of mistakes, and generally thrash around. See if the program does what you want, or at least, gently guides you instead of blowing up. Be impatient. If you can't do what you want right away, give up. If the UI can't withstand your acting generally immature and stupid, it could use some work.

Chapter 9: The Process of Designing a Product

We've talked about the principles of good design, but principles only give you a way to evaluate and improve an existing design. But... how do you figure out what the dang design should be in the first place? Many people write big, functional outlines of all the features they thought up. Then they design each one, and hang it off of a menu item (or web page). When they're done, the program (or web site) has all the functionality they wanted, but it doesn't *flow* right. People sit down and they don't know what it does, and they don't know how to accomplish what they want.

Microsoft's solution to this is something called Activity Based Planning. (As far as I can tell, this concept was invented by Mike Conte on the Excel team, who got bored with that and went on to a second career as a race car driver). The key insight is to figure out the *activity* that the user is doing, and focus on making it easy to accomplish that activity. This is best illustrated with an example.

You've decided to make a web site that lets people create greeting cards. Using a somewhat naïve approach, you might come up with a list of features like this:

1. Add text to card
2. Add picture to card
3. Get predesigned card from library
4. Send card:
    a. Using email
    b. By printing it out

For lack of any better way of thinking about the problem, this might lead itself to a typical Macintosh user interface, circa-1984: a program that starts out with a blank card, with menu items for adding text, pictures, loading cards from a library, and sending cards. And then what the user is going to have to do is sit down and browse through the menus, trying to figure out all the commands available, and then do their own synthesis of how to put these atomic commands together to create a card.

Now, activity based planning says that you need to come up with a list of activities that users might do. So, you talk to your potential users, and you come up with this "top three" list:

1. Birthday Greeting
2. Party Invitation
3. Anniversary Greeting

Now, instead of thinking about your program like a programmer (in terms of *what features you need to have to make a card*), you're thinking about it like the user, in terms of, what activities is the user doing, specifically:

1. Sending a birthday card
2. Planning a party, and inviting people to it
3. Sending an anniversary card

Suddenly, all *kinds* of ideas will rush into your head. Instead of starting with a blank card, you might start with a menu like this:

What do you want to do? • Send a birthday card • Send an anniversary card • Send a party invitation • Start with a blank card

Suddenly users will find it *much* easier to get started with your program, without browsing around on the menus, since the program will virtually lead them through the steps to complete the activity. (There is a risk that if you didn't pick the activities correctly, you will alienate or confuse users who might have been able to use your program, say, to send a Hanukah card, but don't see that as a choice. So be careful in picking activities that blanket the majority of the market you want to target.)

Just looking at our list of three activities suggests some great features which you might want to add. For example, if you're sending a birthday or anniversary card, you might want to be reminded next year to send a card to the same person... so you might add a checkbox that says "remind me next year". And a party invitation needs a way to RSVP, so you might add a feature that lets you collect RSVPs from people electronically. Both of these feature ideas almost fell out of looking at the *activity* that users were performing instead of the *features* in the application.

This example is trivial; for any serious application, the rewards of activity based planning are even greater. When you're designing a program from scratch, you already have a vision of what activities your users are going to be doing. Figuring out this vision is not hard at all, it takes almost no effort at all to do some brainstorming with coworkers, write down a list of potential activities, and then decide which ones you want to focus on. But forcing yourself to list these activities on paper will help your overall design enormously.

Activity based planning is even more important when you are working on version two of a product that people are already using. Here, it may be a matter of observing a sample of customers to see what they are using your program for.

In the days of Excel 1.0 through 4.0, most people at Microsoft thought that the most common user activity was doing financial *what-if* scenarios, where you do things like change the inflation rate and see how this affects your profitability.

When we were designing Excel 5.0, the first major release to use serious activity-based planning, we only had to watch about five customers using the product before we realized that an enormous number of people just use Excel to keep *lists*. They are not entering any formulas or doing any calculation at all! We hadn't even considered this before. Keeping lists turned out to be far more popular than any other activity with Excel. And this led us to invent a whole *slew* of features that make it easier to keep lists: easier sorting, automatic data entry, the AutoFilter feature which helps you see a slice of your list, and multi-user features which let several people work on the same list at the same time while Excel automatically reconciles everything.

While Excel 5 was being designed, Lotus had shipped a "new paradigm" spreadsheet

called Improv. According to the press releases, Improv was a whole new generation of spreadsheet, which was going to blow away everything that existed before it. For various strange reasons, Improv was first available on the NeXT, which certainly didn't help its sales, but a lot of smart people believed that Improv would be to NeXT as VisiCalc was to the Apple II: it would be the *killer app* that made people go out and buy all new hardware just to run one program.

Of course, Improv is now a footnote in history. Search for it on the web, and the only links you'll find are from very over-organized storeroom managers who have, for some reason, made a web site with an inventory of all the stuff they have collecting dust.

Why? Because in Improv, it was almost impossible to just make lists. The Improv designers thought that people were using spreadsheets to create complicated multi-dimensional financial models. Turns out, if they asked people, they would discover that making lists was so much more common than multi-dimensional financial models, and in Improv, making lists was a downright *chore,* if not impossible.

So activity based planning is helpful in the initial version of your application, where you have to make guesses about what people want to do, but it's even more helpful when you're planning the upgrade, because you understand what your customers are doing.

Another example, from the web, is the evolution of deja.com, which started out as an huge, searchable index of Usenet called dejanews. The original interface basically had an edit box and said "search Usenet for *blah*," and that was it. In 1999 a bit of activity based planning showed that one common user activity was doing research on a product or service, of the "which car should I buy" nature. Deja was completely reorganized, and today, it is more of a product opinion research service: the Usenet searching ability is almost completely hidden. This annoyed the small number of users who were using the site to search for whether their Matrox video card worked with Redhat Linux 5.1, but it delighted the much larger population of users who just wanted to buy the best digital camera.

The other great thing about activity based planning is that it lets you make a list of what features *not* to do. When you create *any* kind of software, the reality is that you will come up with three times as many features as you have time to do. And one of the best ways to decide which features get done, and which features get left out, is to evaluate *which features support the most important user activities*.

Imaginary Users.

The very best UI designers in the industry all agree on one thing: you have to invent and describe some imaginary users before you can design your UI. You may remember back in the introduction to this book, I introduced an imaginary user Pete:

> Pete is an accountant for a technical publisher who has used Windows for six years at the office and a bit at home. He is fairly competent and technical. He installs his own software; he reads PC Magazine, and he has even programmed some simple Word macros to help the secretaries in his office send invoices. He's getting a cable modem at home. Pete has

> never used a Macintosh. "They're too expensive," he'll tell you. "You can get a 700 Mhz PC with 128 Meg RAM for the price of..." OK, Pete. We get it.

When you read this, you can almost *imagine* a user. I could also have invented quite another type of user:

> Patricia is an English professor who has written several well-received books of poetry. She has been using computers for word processing since 1980, although the only two programs she ever used are Nota Bene (an ancient academic word processor) and Microsoft Word. She doesn't want to spend time learning the theory of how the computer works, and she tends to store all her documents in whatever directory they would go in if you didn't know about directories.

Obviously, designing software for Pete is quite different from designing software for Patricia, who in turn is quite different from Mike, a 16 year old who runs Linux at home, talks on IRC for hours, and uses no "Micro$oft" software.

When you invent these users, thinking about whether your design is appropriate becomes much easier. For example, a lot of programmers tend to overestimate the ability of the typical user to figure things out. Whenever I write something about command line interfaces being hard to use, I get the inevitable email barrage saying that command line interfaces are ultra-powerful because you can do things like 'gunzip foo.tar.gz | tar xvf -'. But as soon as you have to think about getting Patricia to type "gunzip..." it becomes obvious that that kind of interface just isn't going to serve her needs, ever. Thinking about a "real" person gives you the empathy you need to make a feature that serves that person's need. (Of course, if you're making Linux backup software for advanced sysadmins, you need to invent a character like "Frank" who refuses to touch Windows, which he only refers to as an "operating system" in quotation marks, uses his own personally modified version of tcsh, and runs X11 with four tiled xterms all day long. And about 11 xperfs.)

To summarize, designing good software takes about six steps:

1. Invent some users
2. Figure out the important activities
3. Figure out the *user model* -- how the user will expect to accomplish those activities
4. Sketch out the first draft of the design
5. Iterate over your design again and again, making it easier and easier until it's well within the capabilities of your imaginary users
6. Watch real humans trying to use your software. Note the areas where people have trouble, which probably demonstrate areas where the program model isn't matching the user model.

Good UI sells software, but it also *makes people happy*, because people are happy when they accomplish the task they wanted to accomplish. Which is why UI design is such a satisfying field to be in. Where else are you going to get a chance to make

millions of people just a little bit happier?

Are you a student looking for a great job next summer? Fog Creek Software, a small and friendly startup in New York City, offers summer internships in software development for Computer Science students.

To receive an occasional message when I write a major new article, please subscribe to my spam-free mailing list.
Email: Subscribe

The contents of these pages represent the opinions of one person.
All contents Copyright © 1999-2003 by Joel Spolsky. All Rights Reserved.