About Face

The Essentials of Interaction Design

3

An international bestseller,
now completely revised and updated

Alan Cooper,
Robert Reimann, and David Cronin

11

Eliminating Excise

Software too often contains interactions that are top-heavy, requiring extra work for users. Programmers typically focus so intently on the enabling technology that they don't carefully consider the human actions required to operate the technology from a goal-directed point of view. The result is software that charges its users a tax, or **excise**, of cognitive and physical effort every time it is used.

When we decide to drive to the office, we must open the garage door, get in the car, start the motor, back out, and close the garage door before we even begin the forward motion that will take us to our destination. All these actions support the automobile rather than getting to the destination. If we had *Star Trek* transporters instead, we'd dial up our destination coordinates and appear there instantaneously — no garages, no motors, no traffic lights. Our point is not to complain about the intricacies of driving, but rather to distinguish between two types of actions we take to accomplish our daily tasks.

Any large task, such as driving to the office, involves many smaller tasks. Some of these tasks work directly towards achieving the goal; these are tasks like steering down the road towards your office. **Excise tasks,** on the other hand, don't contribute directly to reaching the goal, but are necessary to accomplishing it just the same. Such tasks include opening and closing the garage door, starting the engine, and stopping at traffic lights, in addition to putting oil and gas in the car and performing periodic maintenance.

Excise is the extra work that satisfies either the needs of our tools or those of outside agents as we try to achieve our objectives. The distinction is sometimes hard to see because we get so used to the excise being part of our tasks. Most of us drive so frequently that differentiating between the act of opening the garage door and the act of driving towards the destination is difficult. Manipulating the garage door is something we do for the car, not for us, and it doesn't move us towards our destination the way the accelerator pedal and steering wheel do. Stopping at red lights is something imposed on us by our society that, again, doesn't help us achieve our true goal. (In this case, it does help us achieve a related goal of arriving *safely* at our offices.)

Software, too, has a pretty clear dividing line between goal-directed tasks and excise tasks. Like automobiles, some software excise tasks are trivial and performing them is no great hardship. On the other hand, some software excise tasks are as obnoxious as fixing a flat tire. Installation leaps to mind here, as do such excise tasks as configuring networks and backing up our files.

The problem with excise tasks is that the effort we expend in doing them doesn't go directly towards accomplishing our goals. Where we can eliminate the need for excise tasks, we make people more effective and productive and improve the usability of a product, ultimately creating a better user experience. As an interaction designer, you should become sensitive to the presence of excise and take steps to eradicate it with the same enthusiasm a doctor would apply to curing an infection. The existence of excise in user interfaces is a primary cause of user dissatisfaction with software-enabled products. It behooves every designer and product manager to be on the lookout for interaction excise in all its forms and to take the time and energy to see that it is *excised* from their products.

DESIGN principle — Eliminate excise wherever possible.

GUI Excise

One of the main criticisms leveled at graphical user interfaces by computer users who are accustomed to command-line systems is that users must expend extra effort manipulating windows and menus to accomplish something. With a command line, users can just type in a command and the computer executes it immediately. With windowing systems, they must open various folders to find the desired file or application before they can launch it. Then, after it appears on the screen, they must stretch and drag the window until it is in the desired location and configuration.

These complaints are well founded. Extra window manipulation tasks like these are, indeed, excise. They don't move a user towards his goal; they are overhead that the applications demand before they deign to assist that person. But everybody knows that GUIs are easier to use than command-line systems. Who is right?

The confusion arises because the real issues are hidden. The command-line interface forces an even more expensive excise budget on users: They must first memorize the commands. Also, a user cannot easily configure his screen to his own personal requirements. The excise of the command-line interface becomes smaller only after a user has invested significant time and effort in learning it.

On the other hand, for a casual or first-time user, the visual explicitness of the GUI helps him navigate and learn what tasks are appropriate and when. The step-by-step nature of the GUI is a great help to users who aren't yet familiar with the task or the system. It also benefits those users who have more than one task to perform and who must use more than one application at a time.

Excise and expert users

Any user willing to learn a command-line interface automatically qualifies as a power user. And any power user of a command-line interface will quickly become a power user of any other type of interface, GUI included. These users will easily learn each nuance of the applications they use. They will start up each application with a clear idea of exactly what they want to do and how they want to do it. To this user, the assistance offered to the casual or first-time user is just in the way.

We must be careful when we eliminate excise. We must not remove it just to suit power users. Similarly, however, we must not force power users to pay the full price for our providing help to new or infrequent users.

Training wheels

One of the areas where software designers can inadvertently introduce significant amounts of excise is in support for first-time or casual users. It is easy to justify adding facilities to a product that will make it easy for newer users to learn how to use it. Unfortunately, these facilities quickly become excise as users become familiar with the product — perpetual intermediates, as discussed in Chapter 3. Facilities added to software for the purpose of training beginners, such as step-by-step wizards, must be easily turned off. Training wheels are rarely needed for extended periods of time, and although they are a boon to beginners, they are a hindrance to advanced users when they are left on permanently.

DESIGN principle Don't weld on training wheels.

"Pure" excise

Many actions are excise of such purity that nobody needs them, from power users to first-timers. These include most hardware-management tasks that the computer could handle itself, like telling an application which COM port to use. Any demands for such information should be struck from user interfaces and replaced with more intelligent application behavior behind the scenes.

Visual excise

Visual excise is the work that a user has to do to decode visual information, such as finding a single item in a list, figuring out where to begin reading on a screen, or determining which elements on it are clickable and which are merely decoration.

Designers sometimes paint themselves into excise corners by relying too heavily on visual metaphors. Visual metaphors such as desktops with telephones, copy machines, staplers, and fax machines — or file cabinets with folders in drawers — are cases in point. These visual metaphors may make it easy to understand the relationships between interface elements and behaviors, but after users learn these fundamentals, managing the metaphor becomes pure excise (for more discussion on the limitations of visual metaphors, see Chapter 13). In addition, the screen space consumed by the images becomes increasingly egregious, particularly in sovereign posture applications (see Chapter 9 for an extensive discussion of the concept of posture). The more we stare at the application from day to day, the more we resent the number of pixels it takes to tell us what we already know. The little telephone that so charmingly told us how to dial on that first day long ago is now a barrier to quick communication.

Users of transient posture applications often require some instruction to use the product effectively. Allocating screen real estate to this effort typically does not contribute to excise in the same way as it does in sovereign applications. Transient posture applications aren't used frequently, so their users need more assistance understanding what the application does and remembering how to control it. For sovereign posture applications, however, the slightest excise becomes agonizing

over time. Another significant source of visual excise is the use of excessively stylized graphics and interface elements (see Figure 11-1). The use of visual style should always be primarily in support of the clear communication of information and interface behavior.

Depending on the application, some amount of ornamentation may also be desirable to create a particular mood, atmosphere, or personality for the product. However, excessive ornamentation can detract from users' effectiveness by forcing them to decode the various visual elements to understand which are controls and critical information and which are mere ornaments. For more about striking the right balance to create effective visual interface designs, see Chapter 14.

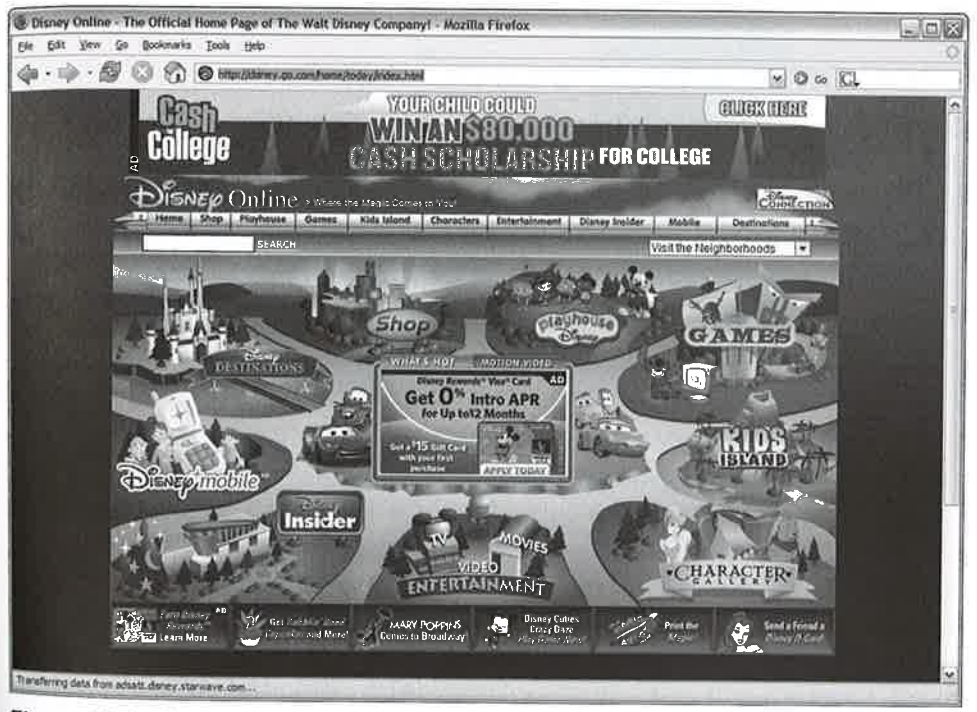

Disney Online - The Official Home Page of The Walt Disney Company! - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Cash College
YOUR CHILD COULD WIN AN $80,000 CASH SCHOLARSHIP FOR COLLEGE
CLICK HERE
Disney Online
Home Shop Playhouse Games Kids Island Characters Entertainment Disney Insider Mobile Destinations
SEARCH
Visit the Neighborhoods
Shop
Playhouse Disney
GAMES
DESTINATIONS
Get 0% Intro APR
Disney mobile
KIDS ISLAND
Insider
MOVIES VIDEO ENTERTAINMENT
CHARACTER GALLERY
Learn More
MARY POPPINS Comes to Broadway
Transferring data from adsatt.disney.starwave.com...

Figure 11-1 The home page at Disney.com provides a good example of visual excise. Text is highly stylized and doesn't follow a layout grid. It's difficult for users to differentiate between décor and navigational elements. This requires users to do visual work to interact with the site. This isn't always a bad thing — just the right amount of the right kind of work can be a good source of entertainment (take puzzles, for example).

Determining what is excise

Certain tasks are mainly excise but can be useful for occasional users or users with special preferences. In this case, consider the function excise if it is forced on a user rather than made available at his discretion. An example of this kind of function is windows management. The only way to determine whether a function or behavior such as this is excise is by comparing it to personas' goals. If a significant persona needs to see two applications at a time on the screen in order to compare or transfer information, the ability to configure the main windows of the applications so that they share the screen space is not excise. If your personas don't have this specific goal, the work required to configure the main window of either application is excise.

Stopping the Proceedings

One form of excise is so prevalent that it deserves special attention. In the previous chapter, we introduced the concept of **flow**, whereby a person enters a highly productive mental state by working in harmony with her tools. Flow is a natural state, and people will enter it without much prodding. It takes some effort to break into flow after someone has achieved it. Interruptions like a ringing telephone will do it, as will an error message box. Some interruptions are unavoidable, but most others are easily dispensable. But interrupting a user's flow for no good reason is *stopping the proceedings with idiocy* and is one of the most disruptive forms of excise.

DESIGN principle Don't stop the proceedings with idiocy.

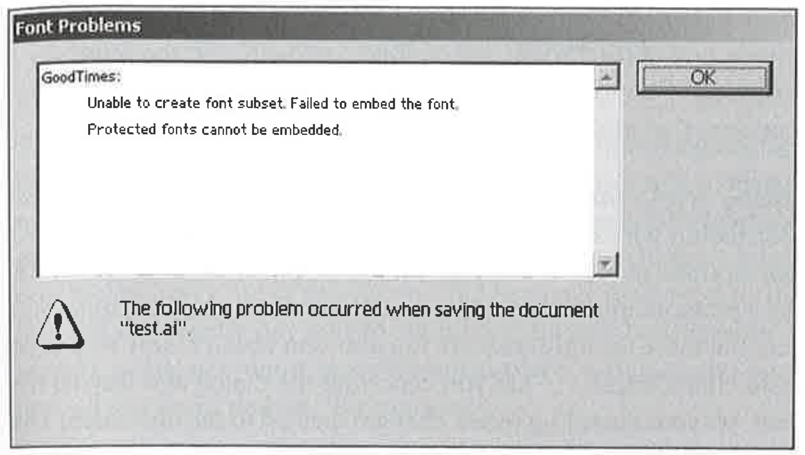
Poorly designed software will make assertions that no self-respecting individual would ever make. It states unequivocally, for example, that a file doesn't exist merely because it is too stupid to look for it in the right place, and then it implicitly blames *you* for losing it! An application cheerfully executes an impossible query that hangs up your system until you decide to reboot. Users view such software behavior as idiocy, and with just cause.

Errors, notifiers, and confirmation messages

There are probably no more prevalent excise elements than error message and confirmation message dialogs. These are so ubiquitous that eradicating them takes a lot of work. In Chapter 25, we discuss these issues at length, but for now, suffice it to

say that they are high in excise and should be eliminated from your applications whenever possible.

The typical error message box is unnecessary. It either tells a user something that he doesn't care about or demands that he fix some situation that the application can and should usually fix just as well. Figure 11-2 shows an error message box displayed by Adobe Illustrator 6 while a user is trying to save a document. We're not exactly sure what it's trying to tell us, but it sounds dire.



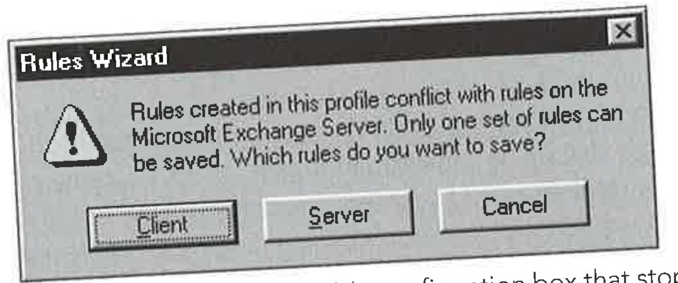
GoodTimes:
Unable to create font subset. Failed to embed the font.
Protected fonts cannot be embedded.

The following problem occurred when saving the document "test.ai".

Figure 11-2 This is an ugly, useless error message box that stops the proceedings with idiocy. You can't verify or identify what it tells you, and it gives you no options for responding other than to admit your own culpability with the OK button. This message comes up only when the application is saving; that is, when you have entrusted it to do something simple and straightforward. The application can't even save a file without help, and it won't even tell you what help it needs.

The message stops an already annoying and time-consuming procedure, making it take even longer. A user cannot reliably fetch a cup of coffee after telling the application to save his artwork, because he might return only to see the function incomplete and the application mindlessly holding up the process. We discuss how to eliminate these sorts of error messages in Chapter 25.

Another frustrating example, this time from Microsoft Outlook, is shown in Figure 11-3.



Rules Wizard

Rules created in this profile conflict with rules on the Microsoft Exchange Server. Only one set of rules can be saved. Which rules do you want to save?

Client | Server | Cancel

Figure 11-3 Here is a horrible confirmation box that stops the proceedings with idiocy. If the application is smart enough to detect the difference, why can't it correct the problem itself? The options the dialog offers are scary. It is telling you that you can explode one of two boxes: one contains garbage, and the other contains the family dog — but the application won't say which is which. And if you click Cancel, what does that mean? Will it still go ahead and explode your dog?

This dialog is asking you to make an irreversible and potentially costly decision based on no information whatsoever! If the dialog occurs just after you changed some rules, doesn't it stand to reason that you want to keep them? And if you don't, wouldn't you like a bit more information, like exactly what rules are in conflict and which of them are the more recently created? You also don't have a clear idea what happens when you click Cancel. . . . Are you canceling the dialog and leaving the rules mismatched? Are you discarding recent changes that led to the mismatch? The kind of fear and uncertainty that this poorly designed interaction arouses in users is completely unnecessary. We discuss how to improve this kind of situation in Chapter 24.

Making users ask permission

Back in the days of command lines and character-based menus, interfaces indirectly offered services to users. If you wanted to change an item, such as your address, first you explicitly had to ask the application permission to change it. The application would then display a screen where your address could be changed. Asking permission is pure excise, and unfortunately things haven't changed much — if you want to change one of your saved addresses on Amazon.com, you have to click a button and go to a different page. If you want to change a displayed value, you should be able to change it right there. You shouldn't have to ask permission or go to a different room.

DESIGN principle — Don't make users ask permission.

As in the last example, many applications have one place where the values (such as filenames, numeric values, and selected options) are displayed for output and another place where user input to them is accepted. This follows the implementation model, which treats input and output as different processes. A user's mental model, however, doesn't recognize a difference. He thinks, "There is the number. I'll just click on it and enter a new value." If the application can't accommodate this impulse, it is needlessly inserting excise into the interface. If options are modifiable by a user, he should be able to do so right where the application displays them.

DESIGN principle — Allow input wherever you have output.

The opposite of asking permission can be useful in certain circumstances. Rather than asking the application to launch a dialog, a user tells a dialog to go away and not come back again. In this way, a user can make an unhelpful dialog box stop badgering him, even though the application mistakenly thinks it is helping. Microsoft now makes heavy use of this idiom. (If a beginner inadvertently dismisses a dialog box and can't figure out how to get it back, he may benefit from another easy-to-identify safety-net idiom in a prominent place: a Help menu item saying, "Bring back all dismissed dialogs," for example.)

Common Excise Traps

You should be vigilant in finding and rooting out each small item of excise in your interface. These myriad little extra unnecessary steps can add up to a lot of extra work for users. This list should help you spot excise transgressions:

- Don't force users to go to another window to perform a function that affects the current window.
- Don't force users to remember where they put things in the hierarchical file system.
- Don't force users to resize windows unnecessarily. When a child window pops up on the screen, the application should size it appropriately for its contents. Don't make it big and empty or so small that it requires constant scrolling.
- Don't force users to move windows. If there is open space on the desktop, put the application there instead of directly over some other already open program.
- Don't force users to reenter their personal settings. If a person has ever set a font, a color, an indentation, or a sound, make sure that she doesn't have to do it again unless she wants a change.

- Don't force users to fill fields to satisfy some arbitrary measure of completeness. If a user wants to omit some details from the transaction entry screen, don't force him to enter them. Assume that he has a good reason for not entering them. The completeness of the database (in most instances) isn't worth badgering users over.
- Don't force users to ask permission. This is frequently a symptom of not allowing input in the same place as output.
- Don't ask users to confirm their actions (this requires a robust undo facility).
- Don't let a user's actions result in an error.

Navigation Is Excise

The most important thing to realize about navigation is that it is largely excise. Except in the case of games where the *goal* is to navigate successfully through a maze of obstacles, the work that users are forced to do to get around in software and on Web sites is seldom aligned with their needs, goals, and desires. (Though it should be noted that well-designed navigation can be an effective way to instruct users about what is available to them, which is certainly much more aligned with their goals.)

Unnecessary or difficult navigation is a major frustration to users. In fact, in our opinion, poorly designed navigation presents one of the largest and most common problems in the usability of interactive products — desktop, Web-based, or otherwise. It is also the place where the programmer's implementation model is typically made most apparent to users.

Navigation through software occurs at multiple levels:

- Among multiple windows, views, or pages
- Among panes or frames within a window, view, or page
- Among tools, commands, or menus
- Within information displayed in a pane or frame (for example: scrolling, panning, zooming, following links)

While you may question the inclusion of some of these bullets as types of navigation, we find it useful to think in terms of a broad definition of navigation: *any action that takes a user to a new part of the interface or which requires him to locate objects, tools, or data.* The reason for this is simple: These actions require people to understand where they are in an interactive system and how to find and actuate what they want. When we start thinking about these actions as navigation, it becomes clear that they are excise and should, therefore, be minimized or eliminated. The following sections discuss each of these types of navigation in more detail.

Navigation among multiple screens, views, or pages

Navigation among multiple application views or Web pages is perhaps the most disorienting kind of navigation for users. It involves a gross shifting of attention that disrupts a user's flow and forces him into a new context. The act of navigating to another window also often means that the contents of the original window are partly or completely obscured. At the very least, it means that a user needs to worry about window management, an excise task that further disrupts his flow. If users must constantly shuttle back and forth between windows to achieve their goals, their disorientation and frustration levels will rise, they will become distracted from the task at hand, and their effectiveness and productivity will drop.

If the number of windows is large enough, a user will become sufficiently disoriented that he may experience **navigational trauma**: He gets lost in the interface. Sovereign posture applications can avoid this problem by placing all main interactions in a single primary view, which may contain multiple independent panes.

Navigation between panes

Windows can contain multiple panes — adjacent to each other and separated by splitters (see Chapters 19 and 20) or stacked on top of each other and denoted by tabs. Adjacent panes can solve many navigation problems because they provide useful supporting functions, links, or data on the screen in close reach of the primary work or display area, thus reducing navigation to almost nil. If objects can be dragged between panes, those panes should be adjacent to each other.

Problems arise when adjacent supporting panes become too numerous or are not placed on the screen in a way that matches users' workflows. Too many adjacent panes result in visual clutter and confusion: Users do not know where to go to find what they need. Also, crowding forces scrolling, which is another navigational hit. Navigation within the single screen thus becomes a problem. Some Web portals, trying to be everything to everyone, have such navigational problems.

In some cases, depending on user workflows, tabbed panes can be appropriate. Tabbed panes include a level of navigational excise and potential for user disorientation because they obscure what was on the screen before the user navigated to them. However, this idiom is appropriate for the main work area when multiple documents or independent views of a document are required (such as in Microsoft Excel; see Figure 11-4).

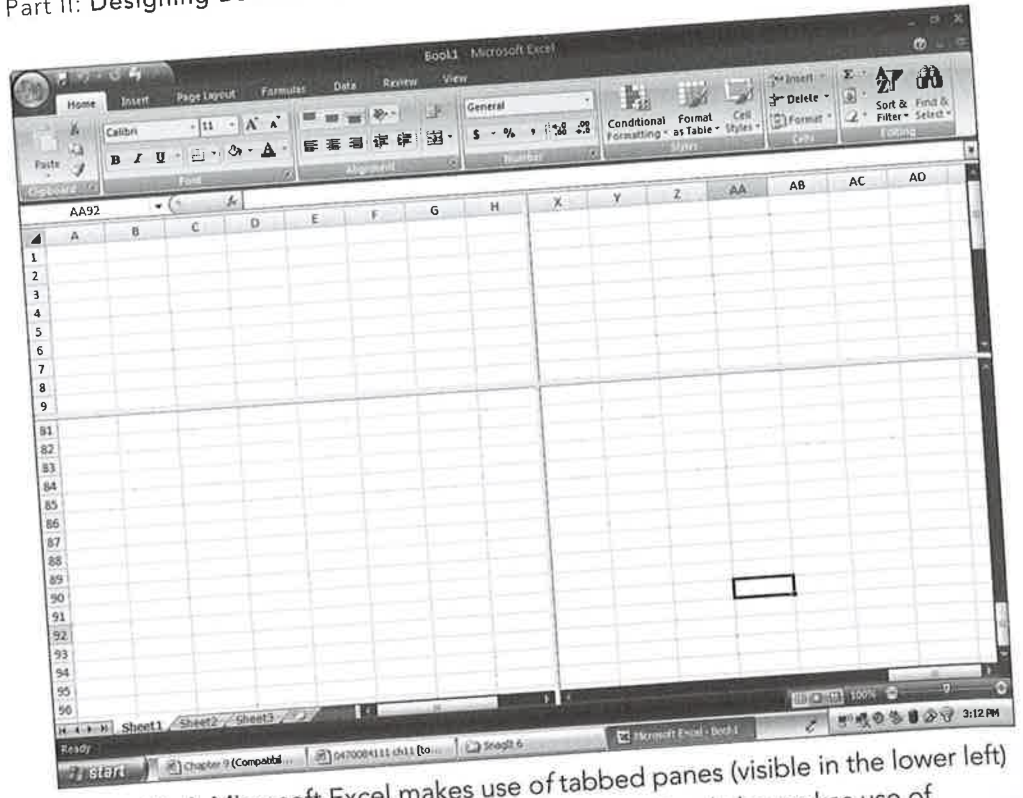


Figure 11-4 Microsoft Excel makes use of tabbed panes (visible in the lower left) to let users navigate between related worksheets. Excel also makes use of splitters to provide adjacent panes for viewing multiple, distant parts of a single spreadsheet without constant scrolling. Both these idioms help reduce navigational excise for Excel users.

Some programmers use tabs to break complex product capabilities into smaller chunks. They reason that using these capabilities will somehow become easier if the functionality is cut into bite-sized pieces. Actually, putting parts of a single facility onto separate panes increases excise and decreases users' understanding and orientation.

The use of tabbed screen areas is a space-saving mechanism and is sometimes necessary to fit all the required information and functions in a limited space. (Settings dialogs are a classic example here. We don't think anyone is interested in seeing all of the settings for a sophisticated application laid bare in a single view.) In most cases, though, the use of tabs creates significant navigational excise. It is rarely possible to describe accurately the contents of a tab with a succinct label. Therefore users must click through each tab to find the tool or piece of information they are looking for.

Tabbed panes can be appropriate when there are multiple supporting panes for a primary work area that are not used at the same time. The support panes can then be stacked, and a user can choose the pane suitable for his current tasks, which is only a single click away. A classic example here involves the color mixer and swatches area in Adobe Illustrator (see Figure 11-5). These two tools are mutually exclusive ways of selecting a drawing color, and users typically know which is appropriate for a given task.

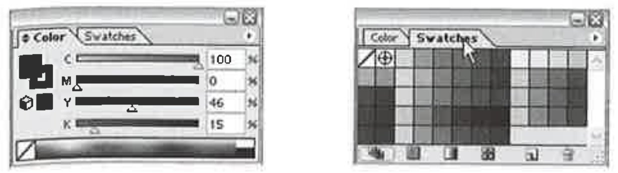


Figure 11-5 Tabbed palettes in Adobe Illustrator allow users to switch between the mixer and swatches, which provide alternate mechanisms for picking a color.

Navigation between tools and menus

Another important and overlooked form of navigation results from users' needs to use different tools, palettes, and functions. Spatial organization of these within a pane or window is critical to minimizing extraneous mouse movements that, at best, could result in user annoyance and fatigue, and at worst, result in repetitive stress injury. Tools that are used frequently and in conjunction with each other should be grouped together spatially and also be immediately available. Menus require more navigational effort on the part of users because their contents are not visible prior to clicking. Frequently used functions should be provided in toolbars, palettes, or the equivalent. Menu use should be reserved only for infrequently accessed commands (we discuss organizing controls again later in this chapter and discuss toolbars in depth in Chapter 23).

Adobe Photoshop 6.0 exhibits some undesirable behaviors in the way it forces users to navigate between palette controls. For example, the Paint Bucket tool and the Gradient tool each occupy the same location on the tool palette; you must select between them by clicking and holding on the visible control, which opens a menu that lets you select between them (shown in Figure 11-6). However, both are fill tools, and both are frequently used. It would have been better to place each of them on the palette next to each other to avoid that frequent, flow-disrupting tool navigation.

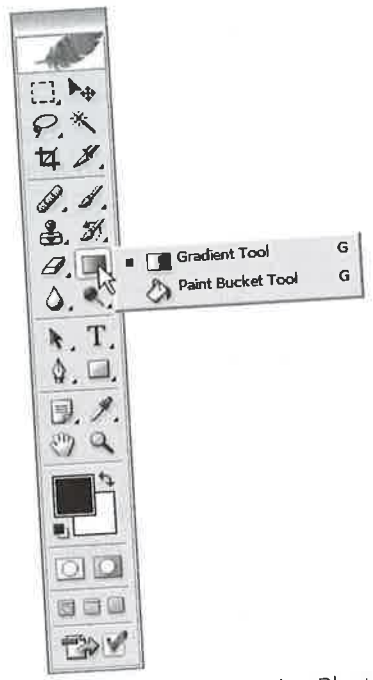


Figure 11-6 In Adobe Photoshop, the Paint Bucket tool is hidden in a combutcon (see Chapter 21) on its tool palette. Even though users make frequent use of both the Gradient tool and the Paint Bucket tool, they are forced to access this menu any time they need to switch between these tools.

Navigation of information

Navigation of information, or of the content of panes or windows, can be accomplished by several methods: scrolling (panning), linking (jumping), and zooming. The first two methods are common: Scrolling is ubiquitous in most software, and linking is ubiquitous on the Web (though increasingly, linking idioms are being adopted in non-Web applications). Zooming is primarily used for visualization of 3D and detailed 2D data.

Scrolling is often a necessity, but the need for it should be minimized when possible. Often there is a trade-off between paging and scrolling information: You should understand your users' mental models and workflows to determine what is best for them.

In 2D visualization and drawing applications, vertical and horizontal scrolling are common. These kinds of interfaces benefit from a thumbnail map to ease navigation. We'll discuss this technique as well as other visual signposts later in this chapter.

Linking is the critical navigational paradigm of the Web. Because it is a visually dislocating activity, extra care must be taken to provide visual and textual cues that help orient users.

Zooming and **panning** are navigational tools for exploring 2D and 3D information. These methods are appropriate when creating 2D or 3D drawings and models or for exploring representations of real-world 3D environments (architectural walkthroughs, for example). They typically fall short when used to examine arbitrary or abstract data presented in more than two dimensions. Some information visualization tools use zoom to mean, "display more attribute details about objects," a logical rather than spatial zoom. As the view of the object enlarges, attributes (often textual) appear superimposed over its graphical representation. This kind of interaction is almost always better served through an adjacent supporting pane that displays the properties of selected objects in a more standard, readable form. Users find spatial zoom difficult enough to understand; logical zoom is arcane to all but visualization researchers and the occasional programmer.

Panning and zooming, especially when paired together, create navigation difficulties for users. While this is improving due to the prevalence of online maps, it is still quite easy for people to get lost in virtual space. Humans are not used to moving in unconstrained 3D space, and they have difficulty perceiving 3D properly when it is projected on a 2D screen (see Chapter 19 for more discussion of 3D manipulation).

Improving Navigation

There are many ways to begin improving (eliminating, reducing, or speeding up) navigation in your applications, Web sites, and devices. Here are the most effective:

- Reduce the number of places to go.
- Provide signposts.
- Provide overviews.
- Provide appropriate mapping of controls to functions.
- Inflect your interface to match user needs.
- Avoid hierarchies.

We'll discuss these in detail below.

Reduce the number of places to go

The most effective method of improving navigation sounds quite obvious: Reduce the number of places to which one must navigate. These "places" include modes, forms, dialogs, pages, windows, and screens. If the number of modes, pages, or screens is kept to a minimum, people's ability to stay oriented increases dramatically. In terms of the four types of navigation presented earlier, this directive means:

- Keep the number of windows and views to a minimum. One full-screen window with two or three views is best for many users. Keep dialogs, especially modeless dialogs, to a minimum. Applications or Web sites with dozens of distinct types of pages, screens, or forms are difficult to navigate.
- Keep the number of adjacent panes in your window or Web page limited to the minimum number needed for users to achieve their goals. In sovereign applications, three panes is a good thing to shoot for, but there are no absolutes here — in fact many applications require more. On Web pages, anything more than two navigation areas and one content area begins to get busy.
- Keep the number of controls limited to as few as your users really need to meet their goals. Having a good grasp of your users via personas will enable you to avoid functions and controls that your users don't really want or need and that, therefore, only get in their way.
- Scrolling should be minimized when possible. This means giving supporting panes enough room to display information so that they don't require constant scrolling. Default views of 2D and 3D diagrams and scenes should be such that a user can orient himself without too much panning around. Zooming, particularly continuous zooming, is the most difficult type of navigation for most users, so its use should be discretionary, not a requirement.

Many online stores present confusing navigation because the designers are trying to serve everyone with one generic site. If a user buys books but never CDs from a site, access to the CD portion of the site could be deemphasized in the main screen for that user. This makes more room for that user to buy books, and the navigation becomes simpler. Conversely, if he visits his account page frequently, his version of the site should have his account button (or tab) presented prominently.

Provide signposts

In addition to reducing the number of navigable places, another way to enhance users' ability to find their way around is by providing better points of reference — **signposts**.

In the same way that sailors navigate by reference to shorelines or stars, users navigate by reference to **persistent objects** placed in a user interface.

Persistent objects, in a desktop world, always include the program's windows. Each application most likely has a main, top-level window. The salient features of that window are also considered persistent objects: menu bars, toolbars, and other palettes or visual features like status bars and rulers. Generally, each window of the interface has a distinctive look that will soon become recognizable.

On the Web, similar rules apply. Well-designed Web sites make careful use of persistent objects that remain constant throughout the shopping experience, especially the top-level navigation bar along the top of the page. Not only do these areas provide clear navigational options, but their consistent presence and layout also help orient customers (see Figure 11-7).

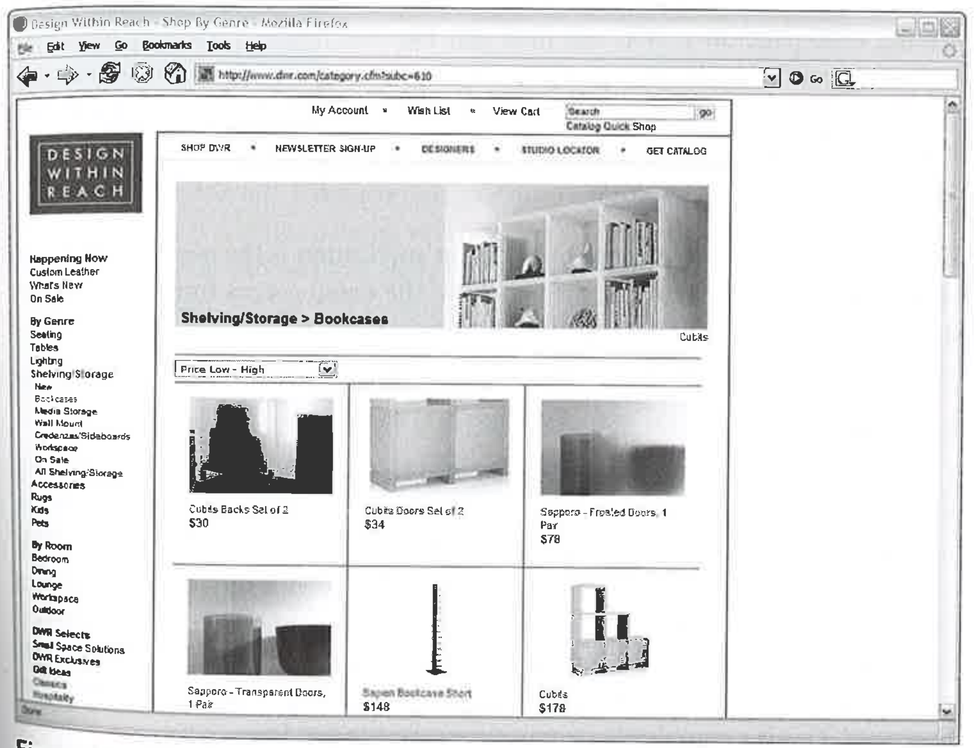


Figure 11-7 The Design Within Reach Web site makes use of many persistent areas on the majority of its pages, such as the links and search field along the top, and the browse tools on the sides. These not only help users figure out where they can go but also help keep them oriented as to where they are.

In devices, similar rules apply to screens, but hardware controls themselves can take on the role of signposts — even more so when they are able to offer visual or tactile feedback about their state. Car radio buttons that, for example, light when selected, even a needle's position on a dial, can provide navigational information if integrated appropriately with the software.

Depending on the application, the contents of the program's main window may also be easily recognizable (especially true in kiosks and small-screen devices). Some applications may offer a few different views of their data, so the overall aspect of their screens will change depending on the view chosen. A desktop application's distinctive look, however, will usually come from its unique combination of menus, palettes, and toolbars. This means that menus and toolbars must be considered aids to navigation. You don't need a lot of signposts to navigate successfully. They just need to be visible. Needless to say, signposts can't aid navigation if they are removed, so it is best if they are permanent fixtures of the interface.

Making each page on a Web site look just like every other one may appeal to marketing, but it can, if carried too far, be disorienting. Certainly, you should use common elements consistently on each page, but by making different rooms look distinct, you will help to orient your users better.

Menus

The most prominent permanent object in an application is the main window and its title and menu bars. Part of the benefit of the menu comes from its reliability and consistency. Unexpected changes to a program's menus can deeply reduce users' trust in them. This is true for menu items as well as for individual menus.

Toolbars

If the application has a toolbar, it should also be considered a recognizable signpost. Because toolbars are idioms for perpetual intermediates rather than for beginners, the strictures against changing menu items don't apply quite as strongly to individual toolbar controls. Removing the toolbar itself is certainly a dislocating change to a persistent object. Although the ability to do so should be there, it shouldn't be offered casually, and users should be protected against accidentally triggering it. Some applications put controls on the toolbar that make the toolbar disappear! This is a completely inappropriate ejector seat lever.

Other interface signposts

Tool palettes and fixed areas of the screen where data is displayed or edited should also be considered persistent objects that add to the navigational ease of the interface. Judicious use of white space and legible fonts is important so that these signposts remain clearly evident and distinct.

Provide overviews

Overviews serve a similar purpose to signposts in an interface: They help to orient users. The difference is that overviews help orient users within the content rather than within the application as a whole. Because of this, the overview area should itself be persistent; its content is dependent on the data being navigated.

Overviews can be graphical or textual, depending on the nature of the content. An excellent example of a graphical overview is the aptly named Navigator palette in Adobe Photoshop (see Figure 11-8).

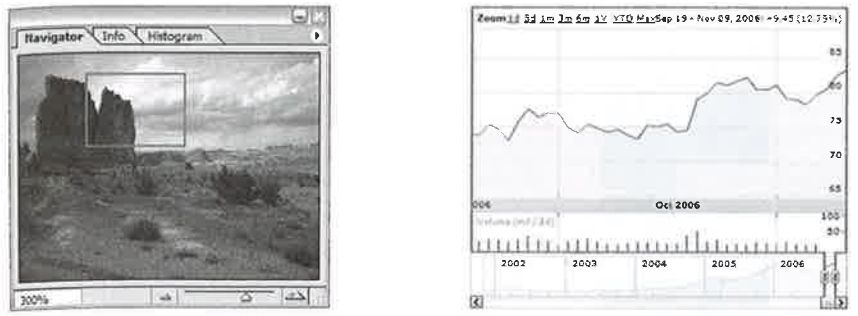
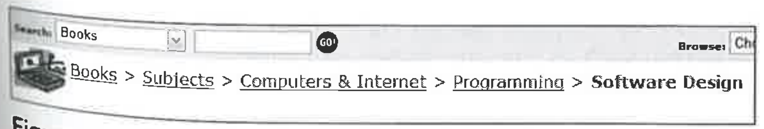


Figure 11-8 On the left, Adobe makes use of an excellent overview idiom in Photoshop: the Navigator palette, which provides a thumbnail view of a large image with an outlined box that represents the portion of the image currently visible in the main display. The palette not only provides navigational context, but it can be used to pan and zoom the main display as well. A similar idiom is employed on the right in the Google Finance charting tool, in which the small graph on the bottom provides a big picture view and context for the zoomed-in view on top.

In the Web world, the most common form of overview area is textual: the ubiquitous breadcrumb display (see Figure 11-9). Again, most breadcrumbs provide not only a navigational aid, but a navigational control as well: They not only show where in the data structure a visitor is, but they give him tools to move to different nodes in the structure in the form of links. This idiom has lost some popularity as Web sites have moved away from strictly hierarchical organizations to more associative organizations, which don't lend themselves as neatly to breadcrumbs.



Books > Subjects > Computers & Internet > Programming > Software Design

Figure 11-9 A typical breadcrumb display from Amazon.com. Users see where they've been and can click anywhere in the breadcrumb trail to navigate to that link.

A final interesting example of an overview tool is the **annotated scrollbar.** Annotated scrollbars are most useful for scrolling through text. They make clever use of the linear nature of both scrollbars and textual information to provide location information about the locations of selections, highlights, and potentially many other attributes of formatted or unformatted text. Hints about the locations of these items appear in the "track" that the thumb of the scrollbar moves in, at the appropriate location. When the thumb is over the annotation, the annotated feature of the text is visible in the display (see Figure 11-10). Microsoft Word uses a variant of the annotated scrollbar; it shows the page number and nearest header in a ToolTip that remains active during the scroll.

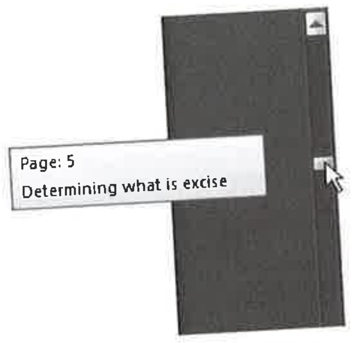


Figure 11-10 An annotated scrollbar from Microsoft Word 2007 provides useful context to a user as he navigates through a document.

Provide appropriate mapping of controls to functions

Mapping describes the relationship between a control, the thing it affects, and the intended result. Poor mapping is evident when a control does not relate visually or symbolically with the object it affects. Poor mapping requires users to stop and think about the relationship, breaking flow. Poor mapping of controls to functions increases the cognitive load for users and can result in potentially serious user errors.

An excellent example of mapping problems comes from the nondigital world of gas and electric ranges. Almost anyone who cooks has run into the annoyance of a stovetop whose burner knobs do not map appropriately to the burners they control. The typical stovetop, such as the one shown in Figure 11-11, features four burners arranged in a flat square with a burner in each corner. However, the knobs that operate those burners are laid out in a straight line on the front of the unit.

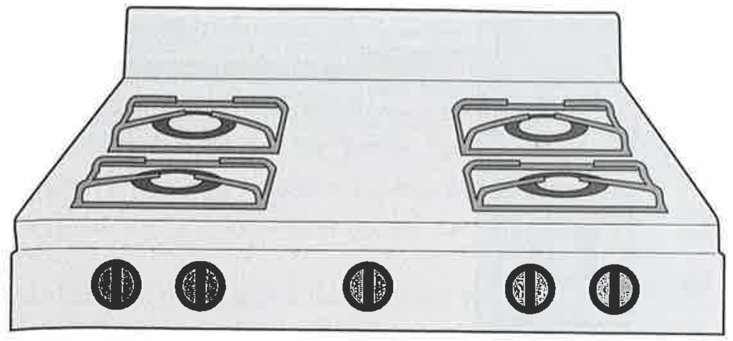


Figure 11-11 A stovetop with poor physical mapping of controls. Does the knob on the far-left control the left-front or left-rear burner? Users must figure out the mapping anew each time they use the stovetop.

In this case, we have a **physical mapping** problem. The *result* of using the control is reasonably clear: A burner will heat up when you turn a knob. However, the *target* of the control — which burner will get warm — is unclear. Does twisting the leftmost knob turn on the left-front burner, or does it turn on the left-rear burner? Users must find out by trial and error or by referring to the tiny icons next to the knobs. The unnaturalness of the mapping compels users to figure this relationship out anew every time they use the stove. This cognitive work may become semiconscious over time, but it still exists, making users prone to error if they are rushed or distracted (as people often are while preparing meals). In the best-case scenario, users feel stupid because they've twisted the wrong knob, and their food doesn't get hot until they notice the error. In the worst-case scenario, they might accidentally burn themselves or set fire to the kitchen.

The solution requires moving the physical locations of the stovetop knobs so that they better suggest which burners they control. The knobs don't have to be laid out in exactly the same pattern as the burners, but they should be positioned so that the target of each knob is clear. The stovetop in Figure 11-12 is a good example of an effective mapping of controls.

In this layout, it's clear that the upper-left knob controls the upper-left burner. The placement of each knob visually suggests which burner it will turn on. Donald Norman (1989) calls this more intuitive layout "natural mapping."

Another example of poor mapping — of a different type — is pictured in Figure 11-13. In this case, it is the **logical mapping** of concepts to actions that is unclear.

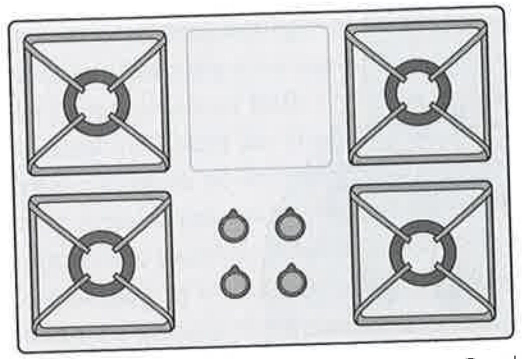


Figure 11-12 Clear spatial mapping. On this stovetop, it is clear which knob maps to which burner because the spatial arrangement of knobs clearly associates each knob with a burner.

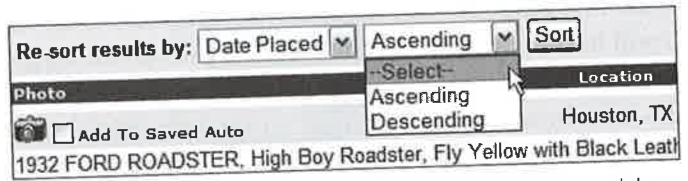

Re-sort results by: Date Placed
Ascending
Sort
--Select--
Ascending
Descending
Photo
Location
Add To Saved Auto
Houston, TX
1932 FORD ROADSTER, High Boy Roadster, Fly Yellow with Black Leatl

Figure 11-13 An example of a logical mapping problem. If a user wants to see the most recent items first, does he choose Ascending or Descending? These terms don't map well to how users conceive of time.

The Web site uses a pair of drop-down menus to sort a list of search results by date. The selection in the first drop-down determines the choices present in the second. When Re-sort Results by: Date Placed is selected in the first menu, the second drop-down presents the options Ascending and Descending.

Unlike the poorly mapped stovetop knobs, the *target* of this control is clear — the drop-down menu selections will affect the list below them. However, the *result* of using the control is unclear: Which sort order will the user get if he chooses Ascending?

The terms chosen to communicate the date sorting options make it unclear what users should choose if they wish to see the most recent items first in the list. Ascending and Descending do not map well to most users' mental model of time. People don't think of dates as ascending or descending; rather, they think of dates and events as being recent or ancient. A quick fix to this problem is to change the wording of the options to Most Recent First and Oldest First, as in Figure 11-14.

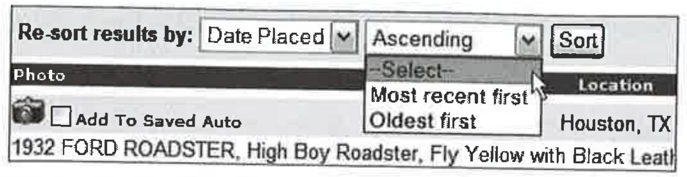

Re-sort results by: Date Placed
Ascending
Sort
--Select--
Most recent first
Oldest first
Photo
Location
Add To Saved Auto
Houston, TX
1932 FORD ROADSTER, High Boy Roadster, Fly Yellow with Black Leatl

Figure 11-14 Clear logical mapping. Most Recent and Oldest are terms that users can easily map to time-based sorting.

Whether you make appliances, desktop applications, or Web sites, your product may have mapping problems. Mapping is an area where attention to detail pays off — you can measurably improve a product by seeking out and fixing mapping problems, even if you have very little time to make changes. The result? A product that is easier to understand and more pleasurable to use.

Inflect your interface to match user needs

Inflecting an interface means organizing it to minimize typical navigation. In practice, this means placing the most frequently desired functions and controls in the most immediate and convenient locations for users to access them, while pushing the less frequently used functions deeper into the interface, where users won't stumble over them. Rarely used facilities shouldn't be removed from the program, but they should be removed from the everyday workspace.

DESIGN principle

Inflect the interface for typical navigation.

The most important principle in the proper inflection of interfaces is **commensurate effort.** Although it applies to all users, it is particularly pertinent to perpetual intermediates. This principle merely states that people will willingly work harder for something that is more valuable to get. The catch, of course, is that value is in the eye of the beholder. It has nothing to do with how technically difficult a feature is to implement, but rather has entirely to do with a person's goals.

If a person really wants something, he will work harder to get it. If someone wants to become a good tennis player, for example, he will get out on the court and play very hard. To someone who doesn't like tennis, any amount of the sport is tedious effort. If a user needs to format beautiful documents with multiple columns, several fonts, and fancy headings to impress his boss, he will be highly motivated to explore the recesses of the application to learn how. He will be putting *commensurate effort* into the project. If some other user just wants to print plain old documents in one column and one font, no amount of inducement will get him to learn those more advanced formatting features.

DESIGN principle

Users make commensurate effort if the rewards justify it.

This means that if you add features to your application that are necessarily complex to manage, users will be willing to tolerate that complexity only if the rewards are worth it. This is why a program's user interface can't be complex to achieve simple results, but it *can* be complex to achieve *complex* results (as long as such results aren't needed very often).

It is acceptable from an interface perspective to make advanced features something that users must expend a little extra effort to activate, whether that means searching in a menu, opening a dialog, or opening a drawer. The principle of commensurate effort allows us to **inflect** interfaces so that simple, commonly used functions are immediately at hand at all times. Advanced features, which are less frequently used but have a big payoff for users, can be safely tucked away where they can be brought up only when needed. Almost any point-and-shoot digital camera serves as a good example of inflection: The most commonly used function — taking a picture — is provided by a prominent button easily accessible at a moment's notice. Less commonly used functions, such as adjusting the exposure, require interaction with onscreen controls.

In general, controls and displays should be organized in an interface according to three attributes: frequency of use, degree of dislocation, and degree of risk exposure.

- **Frequency of use** means how often the controls, functions, objects, or displays are used in typical day-to-day patterns of use. Items and tools that are most frequently used (many times a day) should be immediately in reach, as discussed in Chapter 10. Less frequently used items, used perhaps once or twice a day, should be no more than a click or two away. Other items can be two or three clicks away.
- **Degree of dislocation** refers to the amount of sudden change in an interface or in the document/information being processed by the application caused by the invocation of a specific function or command. Generally speaking, it's a good idea to put these types of functions deeper into the interface (see Chapter 10 for an explanation).

- **Degree of risk exposure** deals with functions that are irreversible or may have other dangerous ramifications. Missiles require two humans turning keys simultaneously on opposite sides of the room to arm them. As with dislocating functions, you want to make these types of functions more difficult for your users to stumble across. The riskiness of an undesirable can even be thought of as a product of the event's likelihood and its ramifications.

Of course, as users get more experienced with these features, they will search for shortcuts, and you must provide them. When software follows commensurate effort, the learning curve doesn't go away, but it disappears from the user's mind — which is just as good.

Avoid hierarchies

Hierarchies are one of the programmer's most durable tools. Much of the data inside applications, along with much of the code that manipulates it, is in hierarchical form. For this reason, many programmers present hierarchies (the implementation model) in user interfaces. Early menus, as we've seen, were hierarchical. But abstract hierarchies are very difficult for users to successfully navigate, except where they're based on user mental models and the categories are truly mutually exclusive. This truth is often difficult for programmers to grasp because they themselves are so comfortable with hierarchies.

Most humans are familiar with hierarchies in their business and family relationships, but hierarchies are not natural concepts for most people when it comes to storing and retrieving arbitrary information. Most mechanical storage systems are simple, composed either of a single sequence of stored objects (like a bookshelf) or a series of sequences, one level deep (like a file cabinet). This method of organizing things into a single layer of groups is extremely common and can be found everywhere in your home and office. Because it never exceeds a single level of nesting, we call this storage paradigm **monocline grouping**.

Programmers are very comfortable with nested systems where an instance of an object is stored in another instance of the same object. Most other humans have a very difficult time with this idea. In the mechanical world, complex storage systems, by necessity, use different mechanical form factors at each level: In a file cabinet, you never see folders inside folders or file drawers inside file drawers. Even the dissimilar nesting of folder-inside-drawer-inside-cabinet rarely exceeds two levels of nesting. In the current desktop metaphor used by most window systems, you can nest folder within folder ad infinitum. It's no wonder most computer neophytes get confused when confronted with this paradigm.

Most people store their papers (and other items) in a series of stacks or piles based on some common characteristic: The Acme papers go here; the Project M papers go there; personal stuff goes in the drawer. Donald Norman (1994) calls this a *pile cabinet*. Only inside computers do people put the Project M documents inside the Active Clients folder, which, in turn, is stored inside the Clients folder, stored inside the Business folder.

Computer science gives us hierarchical structures as tools to solve the very real problems of managing massive quantities of data. But when this implementation model is reflected in the manifest model presented to users (see Chapter 2 for more on these models), they get confused because it conflicts with their mental model of storage systems. Monocline grouping is the mental model people typically bring to the software. Monocline grouping is so dominant outside the computer that interaction designers violate this model at their peril.

Monocline grouping is an inadequate system for physically managing the large quantities of data commonly found on computers, but that doesn't mean it isn't useful as a *manifest model*. The solution to this conundrum is to render the structure as a user imagines it — as monocline grouping — but to provide the search and access tools that only a deep hierarchical organization can offer. In other words, rather than forcing users to navigate deep, complex tree structures, give them tools to *bring appropriate information to them*. We'll discuss some design solutions that help to make this happen in Chapter 15.

12 Designing Good Behavior

As we briefly discussed in Chapter 10, research performed by two Stanford sociologists, Clifford Nass and Byron Reeves, suggests that humans seem to have instincts that tell them how to behave around other sentient beings. As soon as an object exhibits sufficient levels of interactivity — such as that found in your average software application — these instincts are activated. Our reaction to software as sentient is both unconscious and unavoidable.

The implication of this research is profound: If we want users to like our products, we should design them to behave in the same manner as a likeable person. If we want users to be productive with our software, we should design it to behave like a supportive human colleague. To this end, it's useful to consider the appropriate working relationship between human beings and computers.

DESIGN principle The computer does the work and the person does the thinking.

The ideal division of labor in the computer age is very clear: The computer should do the work, and the person should do the thinking. Science fiction writers and computer scientists tantalize us with visions of artificial intelligence: computers that think for themselves. However, humans don't really need much help in